Apache Cassandra

Escalabilidade horizontal para aplicações Java

Casa do Código

OTÁVIO SANTANA

Sumário

- ISBN
- A nova era das aplicações vem com novos bancos de dados
- Sobre o autor
- 1. Conceitos básicos de NoSQL
- 2. Cassandra
- 3. Instalação do Cassandra
- 4. Conhecendo o CQL
- 5. Modelando sua aplicação com Cassandra
- 6. Realizando integração com Java
- 7. Criando um aplicativo com Hibernate
- 8. Criando um aplicativo com Java e Spring
- 9. Criando um aplicativo com Java EE, ops, Jakarta EE
- 10. Considerações finais

ISBN

Impresso e PDF: 978-65-86110-47-0

EPUB: 978-65-86110-44-9

MOBI: 978-65-86110-42-5

Caso você deseje submeter alguma errata ou sugestão, acesse http://erratas.casadocodigo.com.br.

A nova era das aplicações vem com novos bancos de dados

O mundo dos negócios cresce de maneira exponencial com várias oportunidades e empresas concorrendo entre si. A disputa entre clientes é algo que não se define mais pela localização da empresa, mas pela usualidade e, sobretudo, o tempo de resposta. Existem diversos estudos que explicam que melhorias de milissegundos são o suficiente para adquirir novos clientes, fidelizar os já existentes, além de desbancar a concorrência.

Nessa batalha pelos milissegundos, nasceram diversos paradigmas e frameworks, porém, toda essa melhoria seria inútil se a persistência da informação continuasse lenta. Com a necessidade de uma melhor performance, surgiram os bancos de dados NoSQL para facilitar o desempenho e a distribuição da informação.

Junto a esse conceito, nasceu o Apache Cassandra, o banco de dados NoSQL elástico, tolerante a falhas e com um alto grau de performance, tendo cases de sucessos nas maiores empresas do mundo, como o Netflix, GitHub, eBay, dentre outros. O Cassandra é um banco de dados não relacional originado pelo Facebook, e hoje é um banco de dados NoSQL do tipo família de coluna open source dentro da Apache Foundation.

De uma maneira geral, o objetivo do livro é abordar o Cassandra, seus conceitos e sua aplicabilidade com o Java. Para cobrir todos esses conceitos, o livro foi desenhado no total de dez capítulos.

O escopo do inicial é a introdução dos bancos de dados não relacionais. Serão abordados os desafios dos bancos de dados distribuídos, suas limitações com o teorema do CAP, o que é NoSQL e seus tipos, além da comparação entre o NoSQL e o já consolidado banco de dados relacional.

Em seguida, abordaremos os conceitos do Cassandra, como a sua hierarquia, leitura e escrita, seu funcionamento no nó e sua

orquestração dentro de um Cluster. Veremos sua instalação manual, Docker e Cluster com `docker-compose`.

A realização de consultas é muito comum dentro de um banco de dados. No mundo relacional o desenvolvedor está familiarizado com o SQL. O Cassandra tem a sua própria linguagem de consulta: o Cassandra Query Language ou CQL. Você vai aprender como criar keyspace, família de coluna, realizar as operações CRUD (criação, recuperação, atualização e deleção dos dados) que os desenvolvedores tanto amam.

Existem diversas similaridades entre a linguagem da comunicação do Cassandra, o CQL, e o SQL. Vale lembrar que conceitualmente o banco relacional e o Cassandra estão em diferentes espectros, levando em consideração o teorema do CAP. Realizar a modelagem do Cassandra da mesma maneira que se realiza dentro do banco relacional terá um alto impacto de performance.

Após todos os conceitos de comunicação do Cassandra e dicas de modelagem, o próximo passo é a integração com o Java. Dentro disso, o primeiro ponto a ser exibido é o Driver do DataStax, cuja API é similar ao driver JDBC. Um ponto importante é que uma vez o desenvolvedor tenha familiaridade com o CQL toda a comunicação acontece de maneira bastante fluida.

No dia a dia, os negócios se comportam com orientação a objetos, de modo que também é importante que nos debrucemos sobre os impactos dos Mappers, frameworks cujo objetivo é realizar o mapeamento entre o Cassandra e os objetos de uma entidade de negócio, seja de maneira positiva, com a produtividade, ou de maneira negativa, com possíveis problemas de performance. Dentre esses Mappers utilizaremos o DataStax Mapper, Spring Data Cassandra, Hibernate OGM Cassandra e o Eclipse JNoSQL. E para facilitar a comparação entre eles será utilizado o mesmo exemplo que se baseia em uma solução para uma biblioteca.

Para acessar o código-fonte do livro:

https://github.com/otaviojava/cassandra-java-code

Público-alvo/pré-requisitos

Desenvolvedores Java, entusiastas e familiarizados com a tecnologia NoSQL e que desejam aprender ou aprofundar os conhecimentos no Cassandra.

Sobre o autor

Otávio Santana é um engenheiro de software apaixonado com foco em tecnologia Java. Ele tem experiência em persistência poliglota e em aplicações de alta performance em áreas como Finanças, redes sociais e e-commerce. Trabalha em diversas especificações Java como parte do grupo de expert ou como líder da especificação e é membro executivo do Java Community Process. Atua em diversos projetos open sources tanto da Apache como da Eclipse Foundation como Apache Tamaya, Eclipse JNoSQL, Eclipse MicroProfile Jakarta EE, além de ser membro presente da comunidade ajudando os JUG e participar de diversas conferências ao redor do mundo como JavaOne, OracleCode, Devoxx, Qcon, dentre outros.

Em função dos seus esforços para a comunidade e para o open source, recebeu diversos prêmios de reconhecimento como JCP Outstanding Award, Member of the year and innovative JSR, Duke's Choice Award, Java Champion, Oracle GroundBreaker, dentre outros.

Capítulo 1
Conceitos básicos de NoSQL

Os bancos NoSQL são uma classe de tecnologia de persistência que provê um novo mecanismo de armazenamento que vai de encontro com a normalização e os bancos de dados relacionais. Como qualquer outro banco, ele possui os mesmos objetivos: inserir, atualizar, recuperar e deletar informações, porém, com novos conceitos de modelagem e estrutura de armazenamento.

O termo NoSQL, inicialmente, era relacionado com "não SQL" e posteriormente foi estendido para *Not Only SQL*, ou seja, "não apenas SQL", abrindo o conceito de consciência poliglota (o trabalho de lidar com diversos tipos de bancos para alcançar os objetivos na aplicação).

Esses bancos têm como principais características velocidade e a alta taxa de escalabilidade, como facilidade de aumentar a quantidade de servidores de banco de dados. Isso impede o gargalo de operações, evita um ponto de falha, além de distribuí-los geograficamente, fazendo com que os dados estejam próximos dos usuários que farão a requisição.

Bancos NoSQL estão sendo adotados com maior frequência em diversos tipos de aplicações, inclusive para as instituições financeiras. Como consequência, está crescendo também o número de fornecedores para esse tipo de banco de dados.

Atualmente, os bancos de dados NoSQL são classificados em quatro grupos (chave valor, família de coluna, documento e grafos) definidos pelo seu modelo de armazenamento:

Para conhecer um novo conceito, é muito comum realizar uma comparação com o conhecimento já existente. Dessa maneira, considerando que os bancos de dados relacionais são bem famosos, faremos uma comparação entre eles e NoSQL ao decorrer deste capítulo.

1.1 Chave-valor

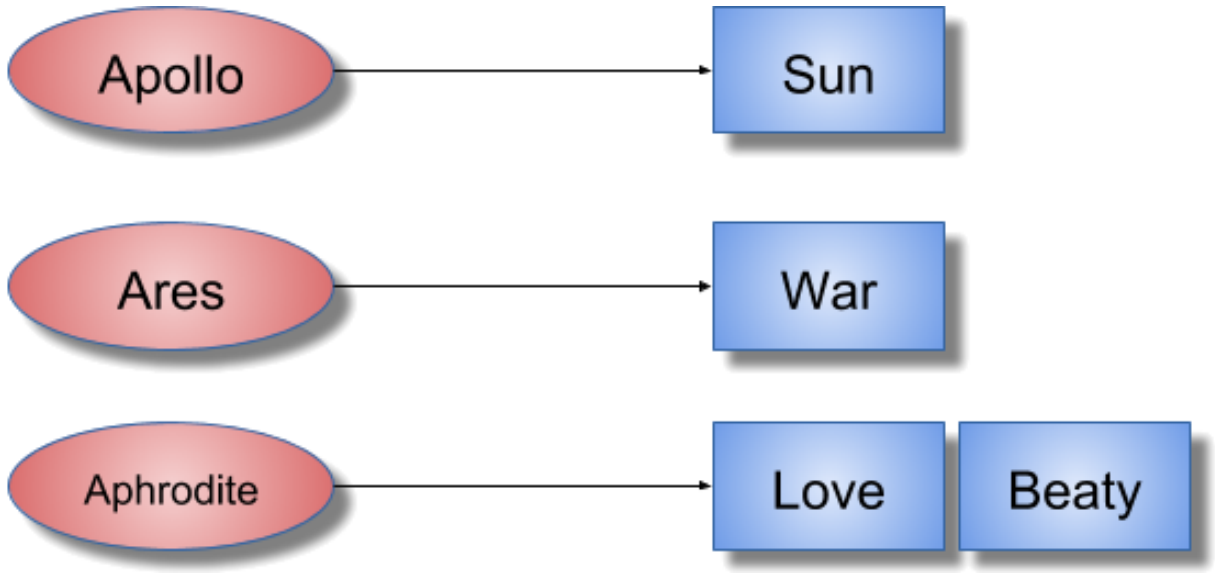

Figura 1.1: Estrutura de chave-valor

Os bancos do tipo chave-valor possuem uma estrutura similar ao `java.util.Map`, ou seja, a informação será recuperada apenas pela chave.

Esse tipo de banco de dados pode ser utilizado, por exemplo, para gerenciar a sessão do usuário. Outro caso interessante é o DNS, cuja chave é o endereço, por exemplo `www.google.com`, e o valor é o IP desse servidor.

Atualmente existem diversas implementações de banco de dados do tipo chave-valor, dentre os quais os mais famosos são:

- AmazonDynamo
- AmazonS3
- Redis
- Scalaris
- Voldemort

Comparando o banco de dados relacional com o do tipo chave-valor, é possível perceber alguns pontos. Um deles é que a estrutura do chave-valor é bastante simples.

Não é possível realizar operações como `join` entre os `buckets` e o valor é composto por grande bloco de informação em vez de ser subdivido em colunas como na base de dados relacional.

Estrutura relacional	**Estrutura chave-valor**
Table	Bucket
Row	Key/value pair
Column	----
Relationship	----

Família de colunas

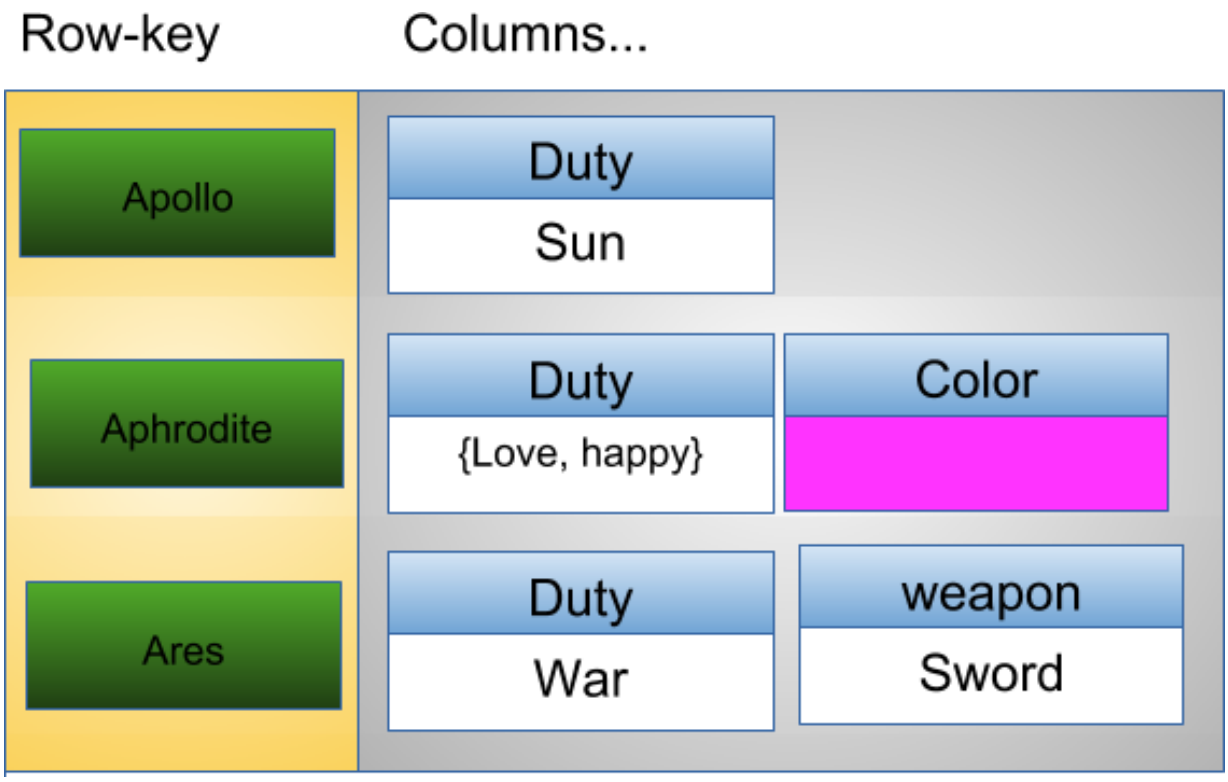

Figura 1.2: Estrutura família de colunas

Esse modelo se tornou popular através do *paper BigTable* do Google, com o objetivo de montar um sistema de armazenamento de dados distribuído, e projetado para ter um alto grau de escalabilidade e de volume de dados. Assim como o chave-valor, para realizar uma busca ou recuperar alguma informação dentro do banco de dados é necessário utilizar o campo que funciona como um identificador único que seria semelhante à chave na estrutura chave-valor. Porém, as semelhanças terminam por aí. As informações são agrupadas em colunas: uma unidade da informação que é composta pelo nome e a informação em si.

Esses tipos de bancos de dados são importantes quando se lidam com um alto grau de volume de dados, de modo que seja necessário distribuir as informações entre diversos servidores. Mas vale salientar que a sua operação de leitura é bastante limitada, semelhante ao chave-valor, pois a busca da informação é definida a

partir de um campo único ou uma chave. Existem diversos bancos de dados que utilizam essas estruturas, por exemplo:

- Hbase
- Cassandra
- Scylla
- Clouddata
- SimpleDb
- DynamoDB

Dentre os tipos de bancos de dados do tipo família de coluna, o Apache Cassandra é o mais famoso. Assim, caso uma aplicação necessite lidar com um grande volume de dados e com fácil escalabilidade, o Cassandra é certamente uma boa opção.

Ao contrapor o banco do tipo família de coluna com os bancos relacionais, é possível perceber que as operações, em geral, são muito mais rápidas. É mais simples trabalhar com grandes volumes de informações e servidores distribuídos em todo o mundo, porém, isso tem um custo: a leitura desse tipo de banco de dados é bem limitada.

Por exemplo, não é possível realizar uniões entre família de colunas como no banco relacional. A família de coluna permite que se tenha um número ilimitado de coluna, que por sua vez é composta por nome e a informação, exatamente como mostra a tabela a seguir:

Estrutura relacional	**Estrutura de família de colunas**
Table	Column Family
Row	Column
Column	nome e valor da Column
Relacionamento	Não tem suporte

1.2 Orientado a documentos

Figura 1.3: Estrutura de documentos

Os bancos de dados orientados a documentos têm sua estrutura muito semelhante a um arquivo JSON ou XML. Eles são compostos por um grande número de campos, que são criados em tempo de

execução, gerando uma grande flexibilidade, tanto para a leitura como para escrita da informação.

Eles permitem que seja realizada a leitura da informação por campos que não sejam a chave. Algumas implementações, por exemplo, têm uma altíssima integração com motores de busca. Assim, esse tipo de banco de dados é crucial quando se realiza análise de dados ou logs de um sistema. Existem algumas implementações dos bancos de dados do tipo documento, sendo que o mais famoso é o MongoDB.

- AmazonSimpleDb
- ApacheCouchdb
- MongoDb
- Riak

Ao comparar com uma base relacional, apesar de ser possível realizar uma busca por campos que não sejam o identificador único, os bancos do tipo documentos não têm suporte a relacionamento. Outro ponto é que os bancos do tipo documento, no geral, são *schemeless*.

Estrutura relacional	**Estrutura de documentos**
Table	Collection
Row	Document
Column	Key/value pair
Relationship	--

1.3 Grafos

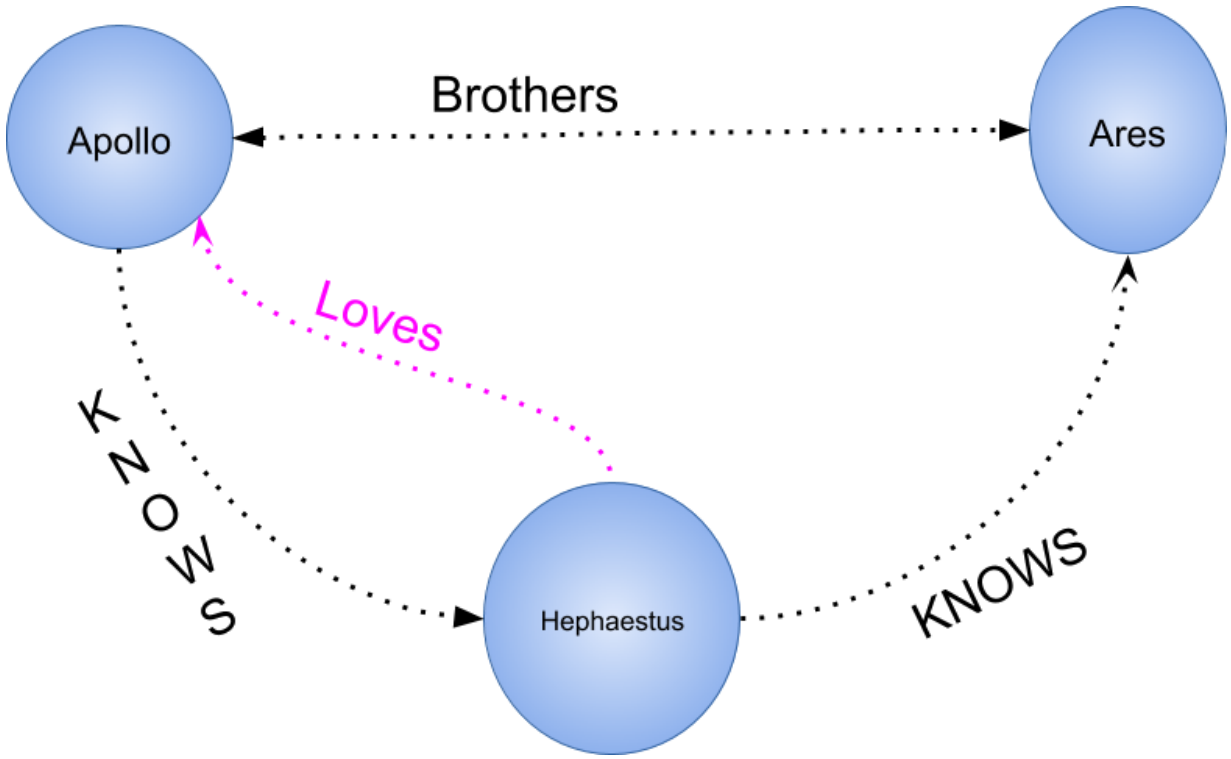

Figura 1.4: Estrutura de grafos

Os bancos do tipo grafos são uma estrutura de dados que conecta um conjunto de vértices através de um conjunto de arestas. Os bancos modernos dessa categoria suportam estruturas de grafo multirrelacionais, onde existem diferentes tipos de vértices (representando pessoas, lugares, itens) e diferentes tipos de arestas. Os sistemas de recomendação que acontecem em redes sociais são o maior case para o banco do tipo grafo. Dos tipos de banco de dados mais famosos no mundo NoSQL, o grafo possui uma estrutura distinta com o relacional.

- Neo4j
- InfoGrid
- Sones
- HyperGraphDB

1.4 Teorema do CAP

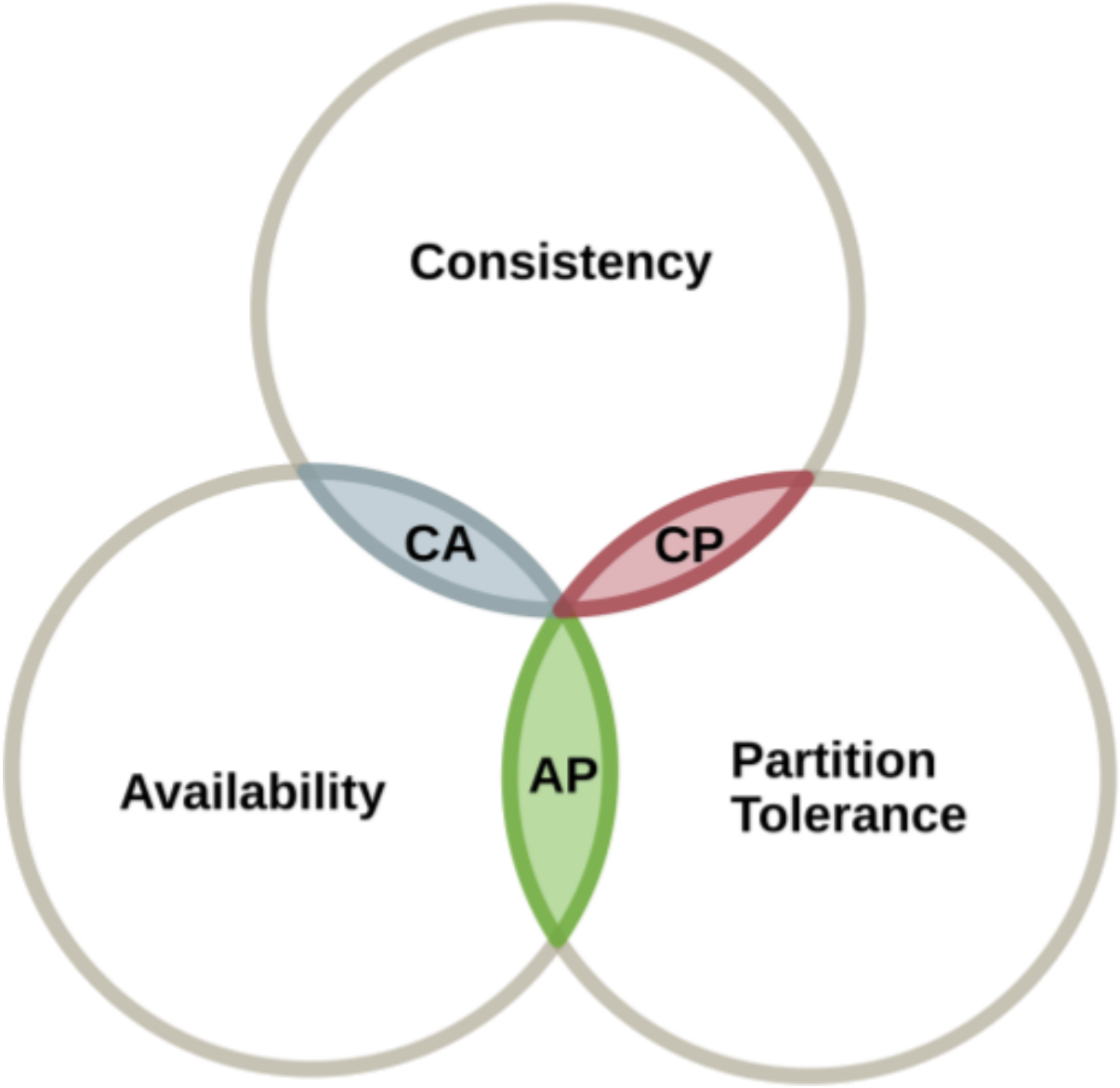

Figura 1.5: Teorema do CAP

Um dos grandes desafios dos bancos de dados NoSQL é que eles lidam com a persistência distribuída, ou seja, as informações ficam localizadas em mais de um servidor. Foram criados diversos estudos para ajudar nesse desafio de persistência distribuída, sendo o mais famoso uma teoria criada em 1999, o Teorema do CAP.

Este teorema afirma que é impossível que o armazenamento de dados distribuído forneça simultaneamente mais de duas das três garantias seguintes:

- *Consistência/Consistency:*: uma garantia de que cada nó em um cluster distribuído retorna a mesma gravação mais recente e bem-sucedida. Consistência refere-se a cada cliente com a mesma visão dos dados.
- *Disponibilidade/Availability*: cada pedido recebe uma resposta (sem erro) - sem garantia de que contém a escrita mais recente.
- *Tolerância à partição/Partition tolerance*: o sistema continua a funcionar e a manter suas garantias de consistência apesar das partições de rede. Os sistemas distribuídos que garantem a tolerância continuam operando mesmo que aconteça alguma falha em um dos nós uma vez que existe, pelo menos, um nó para operar o mesmo trabalho e garantir o funcionamento do sistema.

De uma maneira geral, esse teorema explica que não existe mundo perfeito. Quando se escolhe uma característica, perde-se em outra como consequência. Em um mundo ideal, um banco de dados distribuído conseguiria suportar as três características, porém, na realidade, é importante para o desenvolvedor saber o que ele perderá quando escolher entre um ou outro.

Por exemplo, o Apache Cassandra é AP, ou seja, sua arquitetura focará em tolerância a falha e disponibilidade. Existirão perdas na consistência, assim, em alguns momentos um nó retornará informação desatualizada.

Porém, o Cassandra tem o recurso de nível de consistência, de modo que é possível fazer com que algumas requisições ao banco de dados sejam enviadas a todos os nós ao mesmo tempo, garantindo consistência. Vale ressaltar que fazendo isso ele perderá o A, de *availability* do teorema do CAP.

Conclusão

Este capítulo teve como objetivo dar o pontapé inicial para os bancos de dados não relacionais. Foram discutidos conceitos, os tipos de bancos que existem até o momento e suas estruturas. Com esse novo paradigma de persistência, vêm novas possibilidades e novos desafios para as aplicações.

Esse tipo de banco de dados veio para enfrentar a nova era das aplicações, na qual velocidade ou o menor tempo de resposta possível são um grande diferencial. Com este capítulo introdutório, você está apto para seguir desbravando os bancos não relacionais, com o Cassandra.

Capítulo 2
Cassandra

O Cassandra é um banco de dados NoSQL orientado à família de coluna que nasceu para resolver problemas com aplicações que precisam operar com gigantescas cargas de dados, além de poder escalar com grande facilidade. Ele nasceu no facebook e hoje vem sendo usado intensamente por empresas dos mais variados portes, tais como Netflix, Twitter, Instagram, HP, IBM, dentre muitas outras.

Um fator importante que vale ser citado é a sua crescente adoção inclusive em mercados mais conservadores, em instituições financeiras e agências governamentais, como a NASA. Este capítulo tem como principal objetivo falar do corpo do Cassandra, suas características e seus conceitos.

2.1 Definição

Focando um pouco mais nos aspectos técnicos do Cassandra, é possível destacar alguns pontos:

- **Tolerante a falhas**: os dados são replicados para vários nós de modo que, caso um nó caia, um outro estará pronto para substituí-lo sem *downtime*. Esse recurso é graças à sua característica *masterless*, que permite que todos os nós possuam o mesmo comportamento, ou seja, ao mesmo tempo que escreve também lê informações do banco de dados. Cada nó troca as informações diretamente entre si. Isso acontece em ciclos contínuos, e cada operação dessa é dada a partir de um aperto de mão, *handshake*, em pares. Portanto, caso um cluster possua quatro nós (para exemplificar, imagine que os nomes das máquinas sejam A, B, C e D), no primeiro ciclo o nó A

compartilha informações com B, e C com D; no segundo ciclo, A com C, e D com B, e assim por diante. É a partir desse handshake que o nó descobrirá a localizações dos outros, além das informações do banco de dados em si. Esse protocolo de comunicação que o Cassandra utiliza é o Gossip.

- **Descentralizado**: evitar um calcanhar de Aquiles em uma aplicação é sempre um desafio, que se torna muito maior quando se trata de um banco de dados distribuído. Dados podem ser perdidos, a internet pode ser interrompida ou até mesmo um servidor de banco de dados pode ser destruído fisicamente, dentre outros problemas que podem acontecer - e é por isso que evitar um ponto de falha é algo realmente importante nas aplicações. Para o Cassandra, essa característica vem de modo nativo sem necessidade de realizar nenhuma grande modificação.
- **Elástico**: um número de máquinas pode crescer ou diminuir de maneira linear sem nenhuma interrupção na aplicação. Desse modo, em dias com um alto grau de processamento, por exemplo, na semana do black friday, podem ser adicionados novos servidores e, com o fim desse período de pico, as informações podem retornar ao número de nós originais.

2.2 Hierarquia

Sendo o Cassandra um banco de dados NoSQL do tipo família de colunas, ele seguirá a mesma hierarquia explicada anteriormente para esse tipo de banco não relacional. Acima da família de coluna, existe um item, que é o *keyspace*, de modo que a hierarquia do Cassandra ficará da seguinte forma:

- Keyspace
- Column Family

- Column

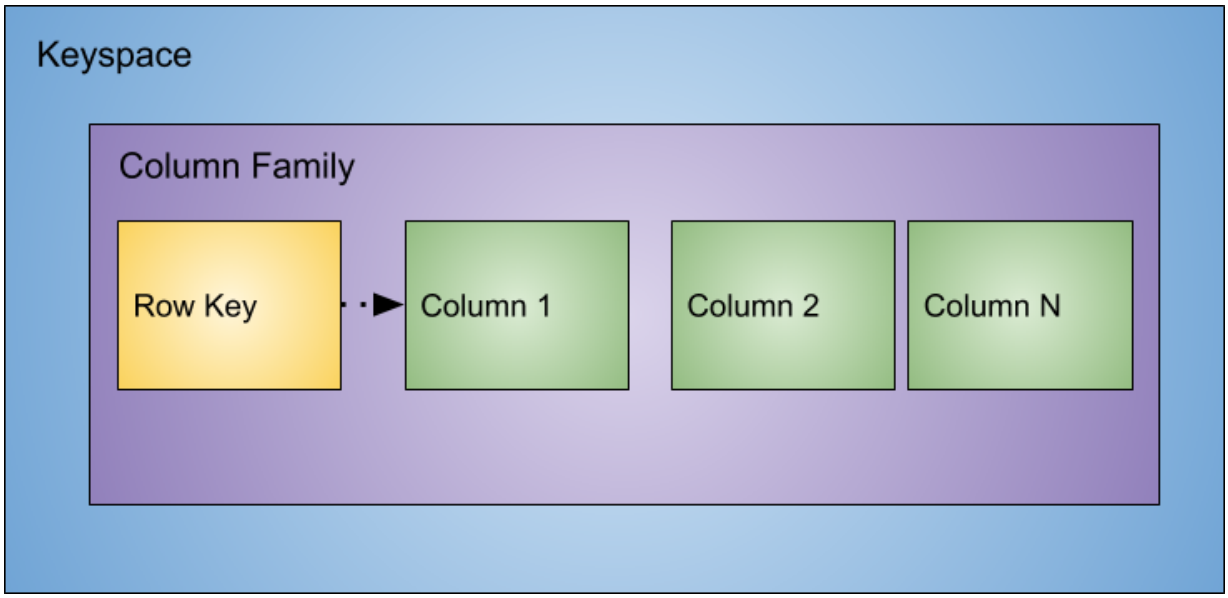

Figura 2.1: A estrutura dentro do Cassandra

Keyspace

O keyspace é a estrutura que armazena ou contém uma ou mais família de colunas. Se pensarmos analogamente a uma tecnologia de dados de persistência relacional, seria semelhante a um banco de dados cuja estrutura armazena as tabelas. Outra responsabilidade igualmente importante que o keyspace possui é a definição do fator de réplica. Ele define o número de vezes que a mesma informação será copiada (replicada) entre os servidores.

Por exemplo, dado um cluster com cinco servidores, e um keyspace com o fator de réplica de quatro, é possível afirmar que nesse cluster a mesma informação será copiada quatro vezes, ou seja, em praticamente todos os nós dentro do data center. Com a criação do fator de réplica vem a metodologia, a definição de como essa informação será copiada entre os nós, chamada de *estratégia de réplica*.

- **SimpleStrategy**: é a estratégia de réplica indicada para caso seja necessário utilizar apenas um único data center com todos

os nós.

- **NetworkTopologyStrategy**: essa estratégia é altamente recomendada para os ambientes de produção e é utilizada quando é necessário utilizar mais de um data center. Isso é recomendado por diversas razões, por exemplo, imagine um desastre natural na região onde um dos data centers se encontram, em São Paulo. Isso não será um problema se o desenvolvedor tiver outros dois data centers na manga, ao redor do Brasil, como em Salvador e outro no Rio de Janeiro. Isso é algo muito semelhante que o Netflix faz com os seus dados com data centers ao redor do mundo.

Column Family e Column

A família de coluna é um contêiner de linhas, sendo que cada linha é composta por uma ou mais colunas. Cada coluna é composta por nome, o valor e o timestamp, que será utilizado para verificar quais informações são as mais atualizadas.

As partes da coluna são:

- **Nome**: que representa o nome da coluna;
- **Valor**: a informação em si que se encontra dentro da coluna;
- **Timestamp**: tão logo seja criada ou alterada, uma coluna tem esse valor atualizado com a data da alteração. Ele serve para informar qual coluna é a mais recente. Por exemplo, durante a comunicação entre dois nós para compartilhar a informação, a partir do protocolo Gossip que informamos anteriormente será esse timestamp que dirá qual coluna é a mais quente, assim, será mantida nos dois bancos de dados.

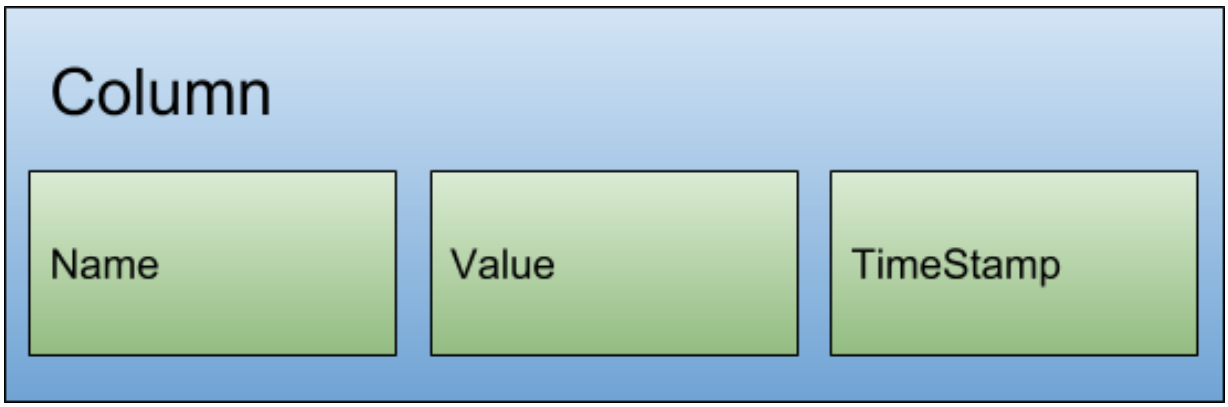

Figura 2.2: A estrutura da Column dentro do Cassandra

2.3 Realizando uma operação dentro do Cassandra

No Cassandra, cada nó tem a responsabilidade tanto de escrita como de leitura. Quando é feita uma requisição para o banco de dados são realizados os seguintes passos:

1. O primeiro é a definição do nó que será responsável por gerenciar o pedido do cliente do banco de dados. Esse nó é conhecido como o coordenador.
2. O nó coordenador será o responsável por operacionar a query entre os demais clusters. Por exemplo, quando se busca uma informação a partir do seu respectivo identificador único, o ID, esse nó gerará valor numérico a partir do ID.
3. Esse valor numérico funcionará como um roteador, uma vez que cada nó é responsável por uma range de valores numéricos, assim, o nó coordenador mandará a requisição para o nó responsável por aquele número.

Uma informação importante é que esse range de valores pelos quais cada nó fica responsável é gerenciado automaticamente de acordo com o número de nós de Cassandra disponíveis.

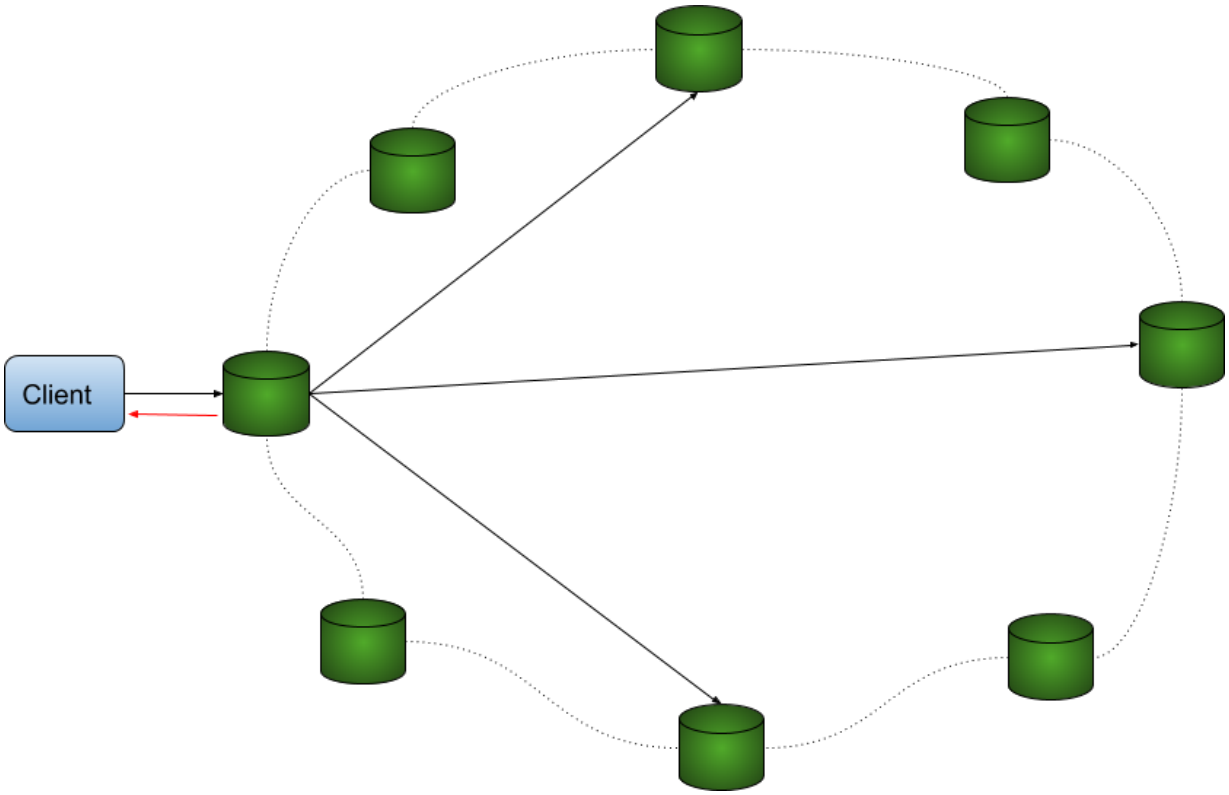

Figura 2.3: A partir das chaves serão gerados um hash, partitioner, com o qual é definido qual nó será responsável pela requisição. Cada nó recebe um range de maneira automática a partir do conceito do Vnode.

O particionador tem a responsabilidade de definir como os dados serão distribuídos ao redor dos nós dentro de um data center. De uma maneira geral, ele gerará um valor numérico a partir do ID. Essa configuração é realizada de maneira global, ou seja, todos os clusters deverão utilizar o mesmo tipo de particionador.

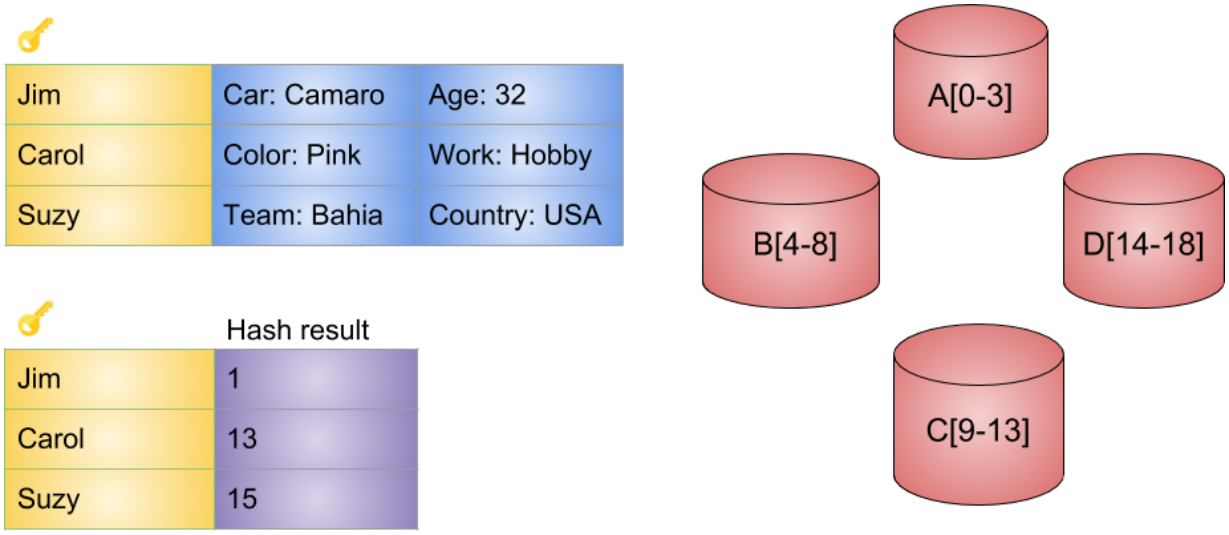

Figura 2.4: A figura exibe a inserção de três registros. Durante uma inserção, é possível perceber que cada registro tem um campo que o define como identificador único ou chave, de cor amarela (Jim, Carol e a Suzy). A partir dessa chave, o partitioner será responsável por gerar um valor numérico e, em seguida, dizer qual nó é o responsável pela aquela informação. No nosso exemplo, a informação com a chave Jim gerou o valor numérico um e irá para o servidor A.

2.4 Consistência versus disponibilidade

Para cada requisição, é possível configurar o nível de consistência com o qual uma operação será realizada.

Esse nível define quantos nós precisam responder ao coordenador para indicar que o processo, de escrita ou leitura, foi realizado com sucesso. Essa configuração é um ponto-chave entre o dilema da consistência e a disponibilidade.

É importante salientar que, quanto maior o número de nós utilizado para uma operação, mais possível é garantir um alto grau de consistência.

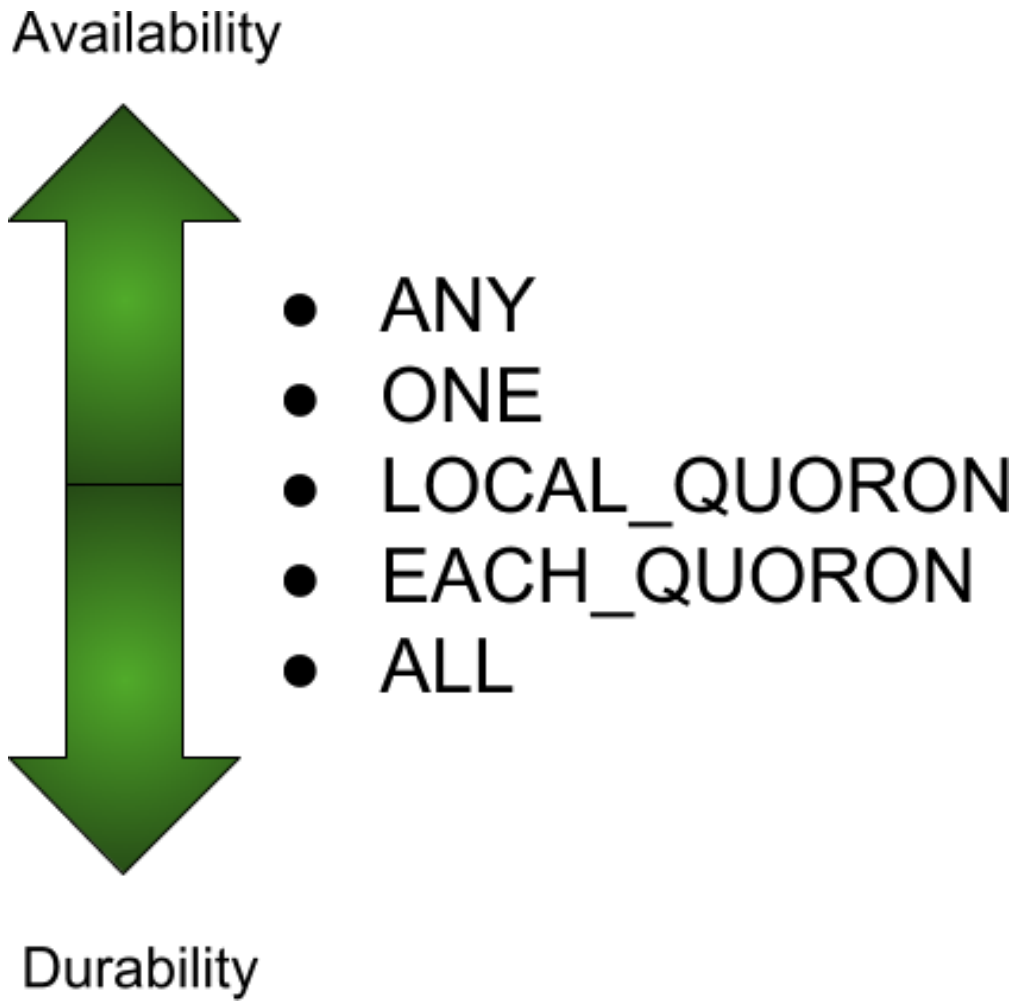

Figura 2.5: Equilíbrio entre consistência e disponibilidade dentro de uma requisição do Cassandra

2.5 Dentro de um nó Cassandra

Uma vez discutido como o Cassandra funciona em cluster, também é preciso falar como ele funciona internamente e de suas partes.

Tão logo o Cassandra recebe uma operação de escrita, ele armazena a informação na memória em uma estrutura chamada de `memtable` e também utiliza uma estrutura no disco, chamada `commit`

`log` . Esse commit log recebe cada escrita feita pelo Cassandra e ela permanece mesmo quando o nó está desligado.

Assim, quando se inicia a escrita, a sequência dentro de um nó Cassandra é:

- Realizar o `logging` dentro do commit log
- Escrever a mesma informação na memória, `memtable`
- Realizar a operação de `flush` a partir do `memtable`
- Armazenar as informações de maneira ordenada dentro do disco com o `SSTables`

Tanto o `memtables` quanto o `SSTable` são armazenados por tabela, de forma organizada e otimizadas para a leitura e escrita, sendo que o `SSTable` tem suas informações dentro do disco.

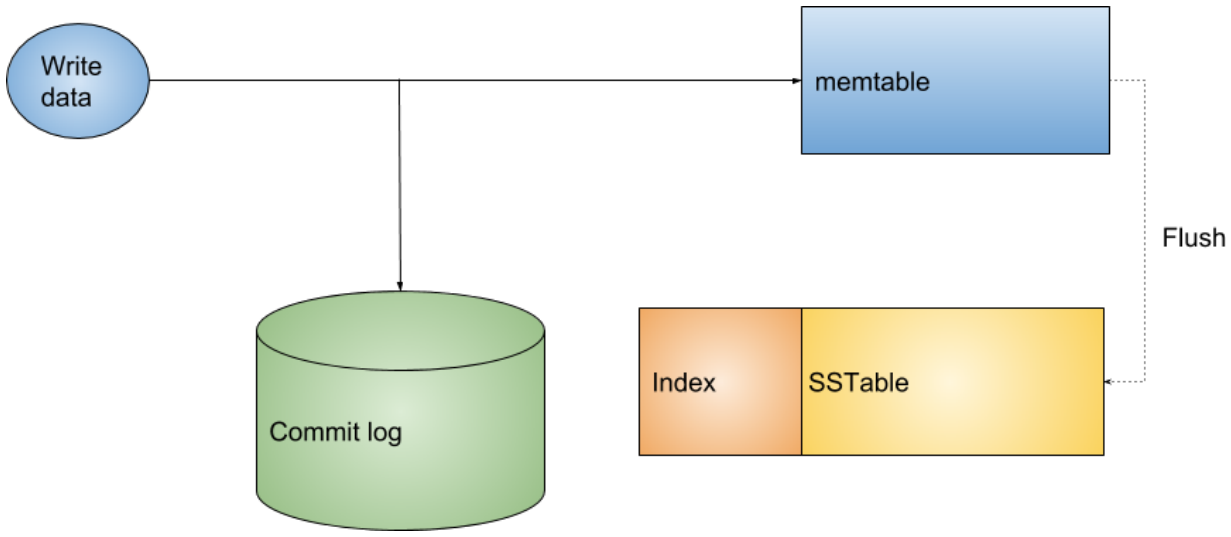

Figura 2.6: A sequência de escrita que acontece dentro de um nó do Cassandra.

Operações de escrita

A tabela a seguir mostra o nível de consistência de escrita do mais forte para o mais fraco, sendo que uma escrita mais forte significa maior consistência, ou seja, escrever em mais nós dentro dos datas centers implica na disponibilidade.

Vale salientar que a operação de escrita em cada nó significa a escrita tanto no `commit log` quanto no `memtable`.

Nível de consistência	Descrição
`ALL`	A operação de escrita em todos os nós da réplica dentro do cluster
`EACH_QUORUM`	A operação de escrita em quorum de réplicas em cada data center
`QUORUM`	Similar ao anterior, escrita em quorum através de todos os data centers
`LOCAL_QUORUM`	A operação de escrita em quorum dentro do data center do nó coordenador
`ONE`	Deve acontecer a operação de escrita em, pelo menos, um nó
`TWO`	Define a operação de escrita em, pelo menos, dois nós
`THREE`	Define a operação de escrita em, pelo menos, três nós
`LOCAL_ONE`	Deve realizar a operação de escrita em, pelo menos, um nó dentro do data center
`ANY`	Garante a operação de escrita em, pelo menos, um nó.

QUORUM é mais da metade, isto é, a metade mais um. A fórmula é `(fator de réplica/2) + 1`

Operações de leitura

A tabela a seguir mostra a informação de leitura do mais forte para o mais fraco levando em consideração a consistência. Um ponto importante é que em cada operação de leitura existe o conceito de *read repair*.

O read repair melhora as consistências dos dados se baseando na simples estratégia de que, após uma requisição, o coordenador pegará as informações mais atualizadas e compartilhará entre todos os nós que participaram da operação de leitura.

Nessa operação, de uma maneira geral, o nó coordenador realiza um *request* e a partir do nível de consistência define o número de nós que participarão da operação. Depois disso, as informações mais “quentes” são enviadas para o cliente, em seguida, é realizada a operação de *read repair* de maneira assíncrona.

Level	Descrição
`ALL`	Retorna a resposta depois de todas as réplicas confirmarem o recebimento dos dados
`QUORUM`	Retorna a resposta depois do quorum de todos os data centers confirmarem o recebimento dos dados
`LOCAL_QUORUM`	Retorna a resposta depois do quorum do data center do nó coordenador
`ONE`	Retorna a resposta depois de ler do nó mais próximo
`TWO`	Retorna a resposta depois de ler dos dois nós mais próximos
`THREE`	Retorna a resposta depois de ler dos três nós mais próximos

Level	Descrição
`LOCAL_ONE`	Retorna a resposta depois de ler do nó mais próximo dentro do data center do nó coordenador
`SERIAL`	Permite a leitura do estado atual (e possivelmente não comprometido) dos dados sem propor uma nova adição ou atualização. Se uma leitura `SERIAL` encontrar uma transação não confirmada em andamento, a transação será confirmada como parte da leitura. Semelhante ao `QUORUM`.
`LOCAL_SERIAL`	Semelhante ao `SERIAL`, porém, confinado ao data center semelhante ao `LOCAL_QUORUM`.

Conclusão

O Apache Cassandra é um banco com uma arquitetura muito intrigante e mostra a plena sintonia do seu funcionamento desde um único nó até o seu funcionamento em conjunto com dezenas ou milhares de nós dentro de diversos data centers ao redor do mundo. Sua forma de distribuir os dados na escrita entre os clusteres a partir da chave faz com que ele foque em disponibilidade e tolerância a falhas. Se pensássemos no teorema do CAP, o Cassandra teria um foco no AP. No próximo capítulo, sairemos um pouco da teoria e aprenderemos como instalar o Cassandra de várias formas.

Capítulo 3
Instalação do Cassandra

Nos capítulos anteriores, foram abordados os conceitos dos bancos de dados não relacionais, suas comparações com o relacional, e dissecamos o funcionamento interno do Cassandra.

Neste capítulo, teremos uma visão mais prática: mostraremos como funciona o processo de instalação, configuração, como criar instâncias do Cassandra dentro um contêiner com Docker, além de como criar um cluster de uma maneira simples com `docker-compose`.

3.1 Realizando download do Cassandra

Antes de iniciar a instalação é importante falar dos pré-requisitos do Cassandra. Como ele é um banco de dados feito em Java, para a instalação é necessário que você tenha instalada alguma implementação do Java 8 (OpenJDK, Azul, Oracle HotSpot etc.).

Nesse primeiro momento de instalação não será utilizado nenhum outro cliente além do `cqlsh`, o comunicador nativo do Cassandra, assim, é necessário que o computador também tenha a última versão do Python 2.7. O `cqlsh` é uma ponte de comunicação com o banco de dados que permite a interação com uma instância via linhas Shell através do CQL, o Cassandra Query Language.

- Realize o download do Cassandra no site do projeto: http://cassandra.apache.org/download/
- Descompacte o arquivo, que terá nome semelhante a `tar -xvf apache-cassandra-3.6-bin.tar.gz`.
- Com os arquivos descompactados, o próximo passo é entrar na pasta `bin` e iniciar o Apache Cassandra. Para isso, execute o comando `cassandra -f`.

Uma vez com a instância de banco de dados de pé, o próximo passo será testar a comunicação com ela. Para isso, será utilizado o `client`, citado anteriormente. Assim, para o executar, é necessário rodar o comando `./cqlsh` dentro da pasta `bin`.

```
Connected to Test Cluster at 127.0.0.1:9042.
[cqlsh 5.0.1 | Cassandra 3.11.3 | CQL spec 3.4.4 | Native protocol v4]
Use HELP for help.
cqlsh> SHOW VERSION
[cqlsh 5.0.1 | Cassandra 3.11.3 | CQL spec 3.4.4 | Native protocol v4]
cqlsh>
```

3.2 Configurações dentro do arquivo yaml

Para executar um único nó, os valores padrões são suficientes, porém, para rodar em um cluster com mais de um nó, algumas mudanças são importantes. As principais configurações se encontram dentro da pasta `conf`, no arquivo `cassandra.yaml`. Destacam-se:

- `cluster_name`: o nome do cluster.
- `seeds`: os IPs, Internet Protocol (o endereço do computador), dos nós sementes separados por vírgulas.
- `listen_address`: os IPs dos nós sementes, ou as instâncias de Cassandra que serão utilizados como referência em uma startup, separados por vírgulas. Essas instâncias terão como responsabilidade “treinar” os novos servidores recém-chegados no cluster. Assim, serão esses nós que serão encarregados de enviar todas as informações necessárias para que o nó calouro consiga trabalhar dentro do cluster.
- `listen_interface`: informa para o Cassandra qual interface utilizar, consequentemente, qual endereço para o uso. É necessário modificar o `listen_address` ou essa configuração, porém, não os dois.

3.3 Simplificando a instalação com contêineres

Uma maneira de se instalar o Cassandra é através de contêiner com o Docker. Em uma visão geral, um contêiner é um ambiente isolado. A tecnologia Docker utiliza o kernel linux e recursos, por exemplo Cgroups e namespaces, para segregar processos, de modo que eles possam ser executados de maneira independente.

O objetivo dos contêineres é criar tal independência: a habilidade de executar diversos processos e aplicativos de maneira isolada, utilizar melhor a infraestrutura, manter a segurança entre os contêineres executados, além da facilidade de criação e manutenção.

> Se você tiver interesse em se aprofundar em Docker, acesse: https://www.casadocodigo.com.br/products/livro-docker

Uma vez instalado o Docker (https://docs.docker.com/install/), basta executar o seguinte comando no console:

```
docker run -d --name casandra-instance -p 9042:9042 cassandra
```

Esse comando baixa e executa https://store.docker.com/images/cassandra, a imagem oficial do Cassandra, diretamente do *docker hub*.

Para executar o `cqlsh` dentro do Docker:

- Liste os contêineres sendo executados na máquina com o comando `docker ps` .

```
$ docker ps
CONTAINER ID        IMAGE
7373093f921a        cassandra
```

- Cada contêiner criado possui um identificador único ou ID. O objetivo do comando anterior foi listar os contêineres existentes

e seus respectivos IDs. Uma vez com o ID do contêiner do Cassandra encontrado, basta executar o comando para `docker exec -it CONTAINER_ID cqlsh` .

```
$ docker exec -it 7373093f921a cqlsh
Connected to Test Cluster at 127.0.0.1:9042.
[cqlsh 5.0.1 | Cassandra 3.11.3 | CQL spec 3.4.4 | Native protocol v4] Use
HELP for help.
cqlsh> SHOW VERSION
[cqlsh 5.0.1 | Cassandra 3.11.3 | CQL spec 3.4.4 | Native protocol v4]
```

Por padrão do Docker, todos os arquivos são criados dentro do contêiner. Assim, para extrair o volume de dados para fora do contêiner, é necessário mapear o caminho `/var/lib/cassandra` , por exemplo:

```
docker run --name some-cassandra -p 9042:9042 -v
/my/own/datadir:/var/lib/cassandra -d cassandra
```

3.4 Criando o primeiro cluster com Docker Compose

Seguindo a linha do Docker e contêiner, para que se execute um cluster é necessário ter muitos contêineres. Umas das ferramentas que permite a execução de múltiplos contêineres é o Docker Compose.

Isso é feito de uma maneira bastante simples com um arquivo YAML de configuração. Dessa forma, com um único comando é possível executar muitos contêineres. O arquivo a seguir mostra uma simples configuração utilizando três nós no cluster.

Toda a descrição de contêineres, configuração de cada um e como eles se inter-relacionam é feita a partir de um arquivo de extensão yml, que por convenção tem como nome `docker-compose.yml` . Esse

arquivo contém a configuração de três nós Cassandra a partir de imagens Dockers.

```
version:  '3.2'

services:

    db-01:
        image: "cassandra"
        networks:
          - cassandranet
        environment:
          broadcast_address: db-01
          seeds: db-01,db-02,db-03
        volumes:
          - /home/otaviojava/Environment/nosql/db1:/var/lib/cassandra

    db-02:
        image: "cassandra"
        networks:
          - cassandranet
        environment:
          broadcast_address: db-02
          seeds: db-01,db-02,db-03
        volumes:
          - /home/otaviojava/Environment/nosql/db2:/var/lib/cassandra

    db-03:
        image: "cassandra"
        networks:
          - cassandranet
        environment:
          broadcast_address: db-03
          seeds: db-01,db-02,db-03
        volumes:
          - /home/otaviojava/Environment/nosql/db3:/var/lib/cassandra

networks:
    cassandranet:
```

Com o arquivo `docker-compose.yml` criado, os próximos passos são muito simples:

1. Para iniciar os contêineres: `docker-compose -f docker-compose.yml up -d`
2. Para parar e remover os contêineres: `docker-compose -f docker-compose.yml down`

> Para esse exemplo, estão sendo levantados três clusters. Caso queria rodar localmente verifique se você terá memória suficiente para isso.

Uma possibilidade é diminuir o consumo de memória dos clusters, por exemplo, iniciando três nós e fazendo com que cada um nó tenha no máximo 1 GB de memória.

Arquivo de configuração de cluster de Cassandra levantando cada nó com 1 gigabyte

```
version:  '3.2'

services:

    db-01:
        image: "cassandra"
        networks:
          - cassandranet
        environment:
          broadcast_address: db-01
          seeds: db-01,db-02,db-03
          JVM_OPTS: -Xms1G -Xmx1G
        volumes:
          - /home/otaviojava/Environment/nosql/db1:/var/lib/cassandra

    db-02:
        image: "cassandra"
        networks:
          - cassandranet
```

```
        environment:
          broadcast_address: db-02
          seeds: db-01,db-02,db-03
          JVM_OPTS: -Xms1G -Xmx1G
        volumes:
          - /home/otaviojava/Environment/nosql/db2:/var/lib/cassandra

    db-03:
        image: "cassandra"
        networks:
          - cassandranet
        environment:
          broadcast_address: db-03
          seeds: db-01,db-02,db-03
          JVM_OPTS: -Xms1G -Xmx1G
        volumes:
          - /home/otaviojava/Environment/nosql/db3:/var/lib/cassandra

networks:
    cassandranet:
```

Conclusão

A instalação e a configuração do Cassandra, seja em cluster utilizando contêiner como Docker ou não, mostraram-se algo realmente muito simples se compararmos a uma configuração de cluster semelhante dentro de um banco de dados relacional.

Um ponto importante é que a popularidade do Docker não é em vão: a sua facilidade de execução e de configuração para um nó ou clusters é realmente muito interessante principalmente para desenvolvedores. Talvez, esse seja o motivo por que atualmente o Docker é considerado a maior ferramenta quando o assunto é DevOps.

Nas próximas cenas, será discutido como realizar a comunicação com o Cassandra, algo que será extremamente simples caso você já esteja acostumado com os bancos relacionais.

Capítulo 4
Conhecendo o CQL

A comunicação é, certamente, a tarefa mais trivial em um banco de dados. O Cassandra oferece sua própria linguagem para comunicação com o banco de dados, o *Cassandra Query language*, ou CQL. O CQL possui uma sintaxe próxima ao SQL, porém, mais simples, já que não existem conceitos como *joins* (*inner join*, *left join*, dentre outros) dentro das buscas do Cassandra. Quem já conhece o SQL terá uma curva de aprendizagem muito menor para aprender essa linguagem do Cassandra.

> Para essas operações utilizaremos o client `cqlsh` que vimos no capítulo anterior.

4.1 Keyspace

A nossa primeira parada no CQL é a criação da maior estrutura dentro da hierarquia do Cassandra, o *keyspace*. É durante a criação do keyspace que é definida a estratégia de replicação e o fator de réplica para os dados. O template da criação é mostrado a seguir:

```
create_keyspace_statement ::=  CREATE KEYSPACE [ IF NOT EXISTS ]
keyspace_name WITH options
```

Por exemplo, ao se criar um keyspace de `library` , livraria em inglês:

```
CREATE KEYSPACE library  WITH replication = {'class': 'SimpleStrategy',
'replication_factor' : 3};
//or
CREATE KEYSPACE library
          WITH replication = {'class': 'NetworkTopologyStrategy', 'DC1' :
```

```
1, 'DC2' : 3}
            AND durable_writes = false;
```

Realizada a criação, é possível verificar os keyspaces criados no Cassandra e para isso será realizada a primeira `query`. É possível perceber a semelhança com o banco de dados relacional. Falaremos mais sobre buscar informações nos itens a seguir.

```
cqlsh> select * from system_schema.keyspaces where keyspace_name=
'library';

 keyspace_name |replication
---------------+-----------------------------------------------------
       library |{'class': 'SimpleStrategy', 'replication_factor': '3'}

(1 rows)
```

Para realizar alguma alteração do keyspace criado, é necessário utilizar o *alter keyspace*. O template é mostrado a seguir:

```
alter_keyspace_statement ::=  ALTER KEYSPACE keyspace_name WITH options
```

Por exemplo, alterando o keyspace criado anteriormente:

```
ALTER KEYSPACE Excelsior WITH replication = {'class': 'SimpleStrategy',
'replication_factor' : 4};
```

Também é possível remover o keyspace com o *drop keyspace*, no nosso exemplo:

```
DROP KEYSPACE library;
```

Caso o usuário tente criar ou remover mais de uma vez a mesma tabela, uma mensagem de erro é gerada. Por exemplo, `Keyspace 'library' already exists` ou `ConfigurationException: Cannot drop non existing keyspace 'library'.` para criar ou remover, respectivamente. Uma maneira de evitar tal erro é realizar essa alteração apenas caso realmente faça sentido: só crie caso ela não exista ou só remova caso ela exista. Com esse objetivo, a sintaxe suporta essa condição:

```
DROP KEYSPACE IF EXISTS library;
CREATE KEYSPACE IF NOT EXISTS library  WITH replication = {'class':
'SimpleStrategy', 'replication_factor' : 3};
```

4.2 Familia de colunas

A criação da família de coluna é semelhante à criação da tabela, ou seja, é nela que são definidos os campos, *partition key*, *clustering key*, o índice e mais configurações. Basta utilizar o template `CREATE TABLE`:

```
create_table_statement ::=  CREATE TABLE [ IF NOT EXISTS ] table_name
                            '('
                                column_definition
                                ( ',' column_definition )*
                                [ ',' PRIMARY KEY '(' primary_key ')' ]
                            ')' [ WITH table_options ]
column_definition      ::=  column_name cql_type [ STATIC ] [ PRIMARY KEY]
primary_key            ::=  partition_key [ ',' clustering_columns ]
partition_key          ::=  column_name
                            | '(' column_name ( ',' column_name )* ')'
clustering_columns     ::=  column_name ( ',' column_name )*
table_options          ::=  COMPACT STORAGE [ AND table_options ]
                            | CLUSTERING ORDER BY '(' clustering_order ')'
[ AND table_options ]
                            | options
clustering_order       ::=  column_name (ASC | DESC) ( ',' column_name
(ASC | DESC) )*
```

Dado que foi criado o keyspace `library`, o próximo passo é criar a família de coluna de livros, `book`:

```
CREATE TABLE library.book (
    name text PRIMARY KEY,
       year int
);
```

É possível também verificar se a família de coluna existe antes de criar, utilizando o `IF NOT EXISTS` semelhante ao keyspace.

Também é possível realizar alterações na família de coluna, por exemplo, adicionar ou remover um campo dentro dela. Para isso, é necessário utilizar o seguinte template:

```
alter_table_statement   ::=  ALTER TABLE table_name
alter_table_instruction
alter_table_instruction ::=  ADD column_name cql_type ( ',' column_name
cql_type )*
                             | DROP column_name ( column_name )*
                             | WITH options
```

Por exemplo, adicionar o campo `author` :

```
ALTER TABLE library.book ADD author text;
```

Caso queria remover o mesmo campo:

```
ALTER TABLE library.book DROP author;
```

Também é possível destruir a estrutura recém-criada.

```
drop_table_statement ::=  DROP TABLE [ IF EXISTS ] table_name
```

No nosso exemplo:

```
DROP TABLE IF EXISTS library.book;
```

Existem casos em que o objetivo não destruir a estrutura, mas remover todo o conteúdo e manter a estrutura. Para isso, é possível truncar a família de coluna, com o comando `TRUNCATE` .

```
truncate_statement ::=  TRUNCATE [ TABLE ] table_name
```

Por exemplo, para criar a família de coluna `book` e inserir dois livros:

```
CREATE TABLE IF NOT EXISTS library.book (
    name text PRIMARY KEY,
       year int
```

```
);
INSERT INTO library.book  JSON '{"name": "Effective Java", "year": 2001}';
INSERT INTO library.book  JSON '{"name": "Clean Code", "year": 2008}';
```

Executando, previamente, a query para buscar os valores existentes no banco de dados:

```
cqlsh> SELECT * FROM library.book;

 name           | year
----------------+------
     Clean Code | 2008
 Effective Java | 2001

(2 rows)
TRUNCATE library.book;
cqlsh> SELECT * FROM library.book;

 name | year
------+------

(0 rows)
```

> É possível utilizar também `COLUMNFAMILY` em vez de `TABLE`.

Chave primária

Dentro da família de coluna, a chave primária (*primary key*) é o campo único e todas as famílias de colunas *devem* defini-la. É a partir desse campo que os dados serão recuperados, por isso, existe uma preocupação inicial com ele. A chave primária pode ser constituída por mais de um campo, porém, ela terá um conceito diferente do banco relacional. Ela será dividida em duas partes:

- O **partition key**: é a primeira parte da chave primária que tem como responsabilidade a distribuição dos dados através dos nós.

- O **clustering columns**: essa chave tem o poder de definir a ordem dentro da tabela. Por exemplo, ao criar uma família de autores definimos o nome como chave e seus livros como ordem.

```
CREATE TABLE IF NOT EXISTS library.author (
    name text,
    book text,
    year int,
    PRIMARY KEY (name, book)
);
INSERT INTO library.author  JSON '{"name": "Robert Cecil Martin", "book":
"Clean Code", "year": 2008}';
INSERT INTO library.author  JSON '{"name": "Robert Cecil Martin", "book":
"Clean Architecture", "year": 2017}';
INSERT INTO library.author  JSON '{"name": "Robert Cecil Martin", "book":
"Agile Principles, Patterns", "year": 2006}';
```

Ao executar a query teremos o seguinte resultado:

```
cqlsh> SELECT * FROM library.author;

 name                | book                         | year
---------------------+------------------------------+------
 Robert Cecil Martin | Agile Principles, Patterns | 2006
 Robert Cecil Martin |         Clean Architecture | 2017
 Robert Cecil Martin |                 Clean Code | 2008

(3 rows)
```

Uma outra opção é remover a tabela e criar novamente, dessa vez com a ordem dos livros de maneira decrescente. Assim:

```
CREATE TABLE IF NOT EXISTS library.author (
    name text,
    book text,
    year int,
    PRIMARY KEY (name, book)
)
WITH CLUSTERING ORDER BY (book DESC);
```

```
INSERT INTO library.author  JSON '{"name": "Robert Cecil Martin", "book":
"Clean Code", "year": 2008}';
INSERT INTO library.author  JSON '{"name": "Robert Cecil Martin", "book":
"Clean Architecture", "year": 2017}';
INSERT INTO library.author  JSON '{"name": "Robert Cecil Martin", "book":
"Agile Principles, Patterns", "year": 2006}';

cqlsh> SELECT * FROM library.author;

 name                | book                         | year
---------------------+------------------------------+------
 Robert Cecil Martin |                  Clean Code | 2008
 Robert Cecil Martin |         Clean Architecture | 2017
 Robert Cecil Martin | Agile Principles, Patterns | 2006

(3 rows)
```

Realizando a busca a partir da chave, nome do autor do livro:

```
cqlsh> SELECT book FROM library.author WHERE name = 'Robert Cecil Martin';

 book
----------------------------
                 Clean Code
         Clean Architecture
 Agile Principles, Patterns

(3 rows)
```

Por padrão, não é possível realizar a busca por um campo que não seja a chave, `partition key` . Assim, no exemplo, tanto a busca pelo livro e pelo ano dará a mesma mensagem de erro.

```
cqlsh> SELECT * FROM library.author WHERE year =2017;
InvalidRequest: Error from server: code=2200 [Invalid query]
message="Cannot execute this query as it might involve data filtering and
thus may have unpredictable performance. If you want to execute this query
despite the performance unpredictability, use ALLOW FILTERING"
cqlsh> SELECT * FROM library.author WHERE book  ='Clean Architecture';
InvalidRequest: Error from server: code=2200 [Invalid query]
message="Cannot execute this query as it might involve data filtering and
```

```
thus may have unpredictable performance. If you want to execute this query
despite the performance unpredictability, use ALLOW FILTERING"
```

Como mostra a mensagem de erro, a única maneira de executar a query é adicionando o comando `ALLOW FILTERING`, porém, isso terá sérias consequências de performance. A maneira como o Cassandra executa esse comando é recuperando todas as linhas e então filtrando por aqueles que não têm o valor da condição. Por exemplo, em uma família de colunas que tenha um milhão de linhas e 98% delas tenha a condição da query, isso será relativamente eficiente. Porém, imagine o caso em que apenas uma linha atenda à condição. O Cassandra terá percorrido de maneira linear e ineficiente 999999 elementos para apenas retornar um.

```
cqlsh> SELECT * FROM library.author WHERE year =2017 ALLOW FILTERING;
 name                | book               | year
---------------------+--------------------+------
 Robert Cecil Martin | Clean Architecture | 2017
(1 rows)

cqlsh> SELECT * FROM library.author WHERE book  ='Clean Architecture'
ALLOW FILTERING;
name                | book               | year
---------------------+--------------------+------
 Robert Cecil Martin | Clean Architecture | 2017

(1 rows)
```

> Se uma query é rejeitada pelo Cassandra pedindo o filtro, resista ao uso do `ALLOW FILTERING`. Verifique sua modelagem, seus dados e volumetria e veja o que você realmente quer fazer. Uma segunda opção é o uso de índices secundários que será abordado a seguir.

Tipos estáticos

Algumas colunas podem ser consideradas estáticas dentro da família de coluna. Uma coluna estática compartilha o mesmo valor para todas as linhas que tenham o mesmo valor de `partition key`. Por exemplo, no cenário de família de coluna autor (com nome e livro), imagine que agora existe o desejo de se adicionar o campo para país de residência. Uma vez que o autor mude de país é importante que todas as suas referências também sejam atualizadas, assim, será criado o campo país, `country`, como estático, exibido a seguir:

Criando a estrutura de `author` agora com o campo estático para país:

```
CREATE TABLE IF NOT EXISTS library.author (
    name text,
    book text,
    year int,
    country text static,
    PRIMARY KEY (name, book)
);
INSERT INTO library.author  JSON '{"name": "Robert Cecil Martin", "book":
"Clean Architecture", "year": 2017}';
INSERT INTO library.author  JSON '{"name": "Robert Cecil Martin", "book":
"Clean Code", "country": "USA", "year": 2008}';
INSERT INTO library.author  JSON '{"name": "Joshua Bloch", "book":
"Effecive Java", "year": 2001}';
INSERT INTO library.author  JSON '{"name": "Joshua Bloch", "book": "JAVA
PUZZLERS", "year": 2005, "country": "Brazil"}';
```

Mesmo inserido uma única vez, ele foi persistido em todos os campos com a mesma chave.

```
cqlsh> SELECT * FROM library.author;

 name                | book               | country | year
---------------------+--------------------+---------+------
        Joshua Bloch |      Effecive Java |  Brazil | 2001
        Joshua Bloch |      JAVA PUZZLERS |  Brazil | 2005
 Robert Cecil Martin | Clean Architecture |     USA | 2017
 Robert Cecil Martin |         Clean Code |     USA | 2008
```

```
(4 rows)
```

Índice secundário

Existe uma outra opção para buscar as informações da tabela além da chave primária, que é o índice secundário. Ele permite a busca de informação sem que gere o erro do `ALLOW FILTERING`. O seu template é exibido a seguir:

```
index_name ::=  re('[a-zA-Z_0-9]+')
```

Por exemplo, para realizar uma busca pelo campo de ano, `year`, na família de coluna de autores é possível executar o seguinte comando:

```
CREATE INDEX year_index ON library.author (year);
```

Assim, finalmente é possível executar a query a partir do ano:

```
cqlsh> SELECT * FROM library.author where year = 2005;

 name         | book           | country | year
--------------+----------------+---------+------
 Joshua Bloch | JAVA PUZZLERS |  Brazil | 2005

(1 rows)
```

Também é possível remover o índice criado com o `DROP INDEX`:

```
drop_index_statement ::=  DROP INDEX [ IF EXISTS ] index_name
```

No nosso exemplo:

```
DROP  INDEX IF EXISTS library.year_index;
```

Como funciona o índice secundário

Os índices secundários são criados para uma coluna dentro de uma família de coluna e são armazenados localmente para cada nó.

Se uma query é baseada em índice secundário, ela não terá a mesma eficiência como a busca pela chave e pode impactar fortemente a performance. Esse é um dos motivos pelos quais algumas documentações do Cassandra considera o uso de índice secundário como anti-pattern, de modo que é importante analisar o impacto da criação de um índice secundário.

> Uma maneira eficiente de usar o índice secundário é em parceria com a chave primária, `partition key` , assim ele será endereçado em um range de nós específicos.

Existem algumas boas práticas ao se utilizar índices, que são:

- **Não** utilizar quando existe um alto grau de cardinalidade;
- **Não** utilizar em campos que são atualizados com uma alta frequência.

4.3 Tipos no Cassandra

Dentro de uma família de colunas cada campo tem um tipo que define como o campo será armazenado no banco de dados. Para facilitar o entendimento, eles serão divididos em quatro:

- Tipos nativos
- Tipos de coleção
- Tuplas
- User-defined-type, UDT

Tipos nativos

Os tipos nativos são aqueles aos quais o Cassandra já tem suporte e não é necessário realizar nenhuma modificação ou criação dentro do banco de dados.

Tipo	descrição
ascii	ASCII String
bigint	Um long de 64-bit
blob	Um grande array de bytes
boolean	Tem o valor true ou falso
counter	Uma coluna contadora (long 64-bit)
date	Uma data
decimal	Um tipo de precisão decimal
double	64-bit IEEE-754 de ponto flutuante
duration	Um tipo de duração com precisão em nanossegundos
float	32-bit IEEE-754 de ponto flutuante
inet	Um endereço IP
int	32-bit int
smallint	16-bit int
text	Uma String como UTF8
time	Uma data com precisão em nanossegundos
timestamp	Um timestamp com precisão em milissegundos
uuid	único identificador universal
varchar	Uma String em UTF8

Tipos de coleção

Dentro do Cassandra existe suporte para três tipos de coleções que seguem a mesma linha do mundo Java. Essas coleções se

encaixam perfeitamente quando é necessário ter um campo que possui conjunto de itens, por exemplo, a lista de telefones ou de contatos.

```
collection_type ::=  MAP '<' cql_type ',' cql_type '>'
                     | SET '<' cql_type '>'
                     | LIST '<' cql_type '>'
```

List

É a uma sequência de itens na ordem que em que foram adicionados. Por exemplo, dada uma família de coluna de leitores é possível ter uma lista livros. Assim:

Criação da família de coluna de leitores

```
CREATE TABLE IF NOT EXISTS library.reader (
    name text,
    books list<text>,
    PRIMARY KEY (name)
);

INSERT INTO library.reader (name, books) VALUES ('Poliana', ['The
Shack','The Love','Clean Code']);
INSERT INTO library.reader (name, books) VALUES ('David', ['Clean
Code','Effectove Java','Clean Code']);

cqlsh> SELECT * FROM library.reader;

 name    | books
---------+-----------------------------------------------
   David | ['Clean Code', 'Effectove Java', 'Clean Code']
 Poliana |        ['The Shack', 'The Love', 'Clean Code']

(2 rows)
```

Dentro da lista é possível realizar diversas opções, por exemplo, adicionar um ou mais elementos, substituir um único item, como mostra o código:

```
//repleace
cqlsh> UPDATE library.reader SET books = [ 'Java EE 8'] WHERE name =
'David';
cqlsh> SELECT * FROM library.reader where name = 'David';

 name    | books
---------+-----------------------------------------
   David |                            ['Java EE 8']

//appending
UPDATE library.reader SET books = books + [ 'Clean Code'] WHERE name =
'David';
cqlsh> SELECT * FROM library.reader where name = 'David';
 name  | books
-------+----------------------------
 David | ['Java EE 8', 'Clean Code']

//update an element
UPDATE library.reader SET books[1]= 'Clean Architecture' WHERE name =
'David';
cqlsh> SELECT * FROM library.reader where name = 'David';
 name  | books
-------+-----------------------------------
 David | ['Java EE 8', 'Clean Architecture']
```

Também é possível remover os elementos com o `DELETE` e com o `UPDATE`.

```
DELETE books[1] FROM library.reader where name = 'David';
cqlsh> SELECT * FROM library.reader WHERE name= 'David';

 name  | books
-------+--------------
 David | ['Java EE 8']


UPDATE library.reader SET books = books - [ 'Java EE 8' ] WHERE name =
'David';
cqlsh> SELECT * FROM library.reader WHERE name= 'David';
 name  | books
```

```
-------+-------
 David |  null
```

Set

O Set é similar ao List, porém, ele não permite valores duplicados. Assim, o mesmo exemplo citado anteriormente, porém com Set, ficaria:

Criação da família de coluna de leitores (dessa vez, é possível ver que o livro “Clean Code” do leitor “David” não aparecerá).

```
DROP TABLE IF EXISTS library.reader;

CREATE TABLE IF NOT EXISTS library.reader (
    name text,
    books set<text>,
    PRIMARY KEY (name)
);

INSERT INTO library.reader (name, books) VALUES ('Poliana', {'The
Shack','The Love','Clean Code'});
INSERT INTO library.reader (name, books) VALUES ('David', {'Clean
Code','Effectove Java','Clean Code'});

cqlsh> SELECT * FROM library.reader;

 name    | books
---------+----------------------------------------
   David |        {'Clean Code', 'Effectove Java'}
 Poliana | {'Clean Code', 'The Love', 'The Shack'}

(2 rows)
```

Dentro do Set é possível tanto substituir a corrente coleção como adicionar novos elementos:

```
//repleace
cqlsh> UPDATE library.reader SET books = { 'Java EE 8'} WHERE name =
'David';
cqlsh> SELECT * FROM library.reader where name = 'David';
```

```
 name    | books
---------+----------------------------------------
   David |                            {'Java EE 8'}

//appending
UPDATE library.reader SET books = books + { 'Clean Code'} WHERE name =
'David';
cqlsh> SELECT * FROM library.reader where name = 'David';
 name  | books
-------+----------------------------
 David | {'Java EE 8', 'Clean Code'}
```

> Como recomendação, o tipo `List` tem algumas limitações e problemas de performance comparado ao `Set` . Caso possa escolher, sempre priorize o `Set` .

Map

O mapa segue a linha de um dicionário de dados ou de um `java.util.Map` , caso o leitor seja do mundo Java. Dado o exemplo anterior de leitores, vamos supor que se deseje adicionar informações de contato. Veja o código a seguir.

```
DROP TABLE IF EXISTS library.reader;

CREATE TABLE IF NOT EXISTS library.reader (
    name text,
    books set<text>,
    contacts map<text,text>,
    PRIMARY KEY (name)
);
INSERT INTO library.reader (name, books, contacts) VALUES ('Poliana',
{'The Shack','The Love','Clean Code'},
{'email': 'poliana@email.com','phone': '+1 55 486848635', 'twitter':
'polianatwitter', 'facebook': 'polianafacebook'});
INSERT INTO library.reader (name, books, contacts) VALUES ('David',
{'Clean Code'}, {'email': 'david@email.com'
,'phone': '+1 55 48684865', 'twitter': 'davidtwitter'});
```

```
cqlsh> SELECT * FROM library.reader;

 name    | books                                        | contacts
---------+----------------------------------------------+----------------------
-------------------------------------------------------------------------------
------------------------
   David |                            {'Clean Code'} |
{'email': 'david@email.com', 'phone': '+1 55 48684865', 'twitter':
'davidtwitter'}
 Poliana | {'Clean Code', 'The Love', 'The Shack'} | {'email':
'poliana@email.com', 'facebook': 'polianafacebook', 'phone': '+1 55
486848635', 'twitter': 'polianatwitter'}
```

Assim como as coleções anteriores, é possível realizar alterações:

```
//to update a key element
UPDATE library.reader SET contacts['twitter'] = 'fakeaccount' WHERE name =
'Poliana';
cqlsh> SELECT * from library.reader where name = 'Poliana';

 name    | books                                        | contacts
---------+----------------------------------------------+----------------------
-------------------------------------------------------------------------------
---------------------
 Poliana | {'Clean Code', 'The Love', 'The Shack'} | {'email':
'poliana@email.com', 'facebook': 'polianafacebook', 'phone': '+1 55
486848635', 'twitter': 'fakeaccount'}
//to append a new entry
UPDATE library.reader SET contacts = contacts+
{'youtube':'youtubeaccount'} WHERE name = 'Poliana';
cqlsh> SELECT * from library.reader where name = 'Poliana';
 name    | books                                        | contacts
---------+----------------------------------------------+----------------------
-------------------------------------------------------------------------------
-------------------------------------------------
 Poliana | {'Clean Code', 'The Love', 'The Shack'} | {'email':
'poliana@email.com', 'facebook': 'polianafacebook', 'phone': '+1 55
486848635', 'twitter': 'fakeaccount', 'youtube': 'youtubeaccount'}
//remove an element
DELETE contacts['youtube'] FROM  library.reader where  name = 'Poliana';
```

```
cqlsh> SELECT * from library.reader where name = 'Poliana';
 name    | books                                        | contacts
---------+----------------------------------------------+----------------------
-------------------------------------------------------------------------------
--------------------
 Poliana | {'Clean Code', 'The Love', 'The Shack'} | {'email':
'poliana@email.com', 'facebook': 'polianafacebook', 'phone': '+1 55
486848635', 'twitter': 'fakeaccount'}
//to remove elements by the key
UPDATE library.reader SET contacts = contacts - {'youtube','twitter'}
WHERE name = 'Poliana';
cqlsh> SELECT * from library.reader where name = 'Poliana';
 name    | books                                        | contacts
---------+----------------------------------------------+----------------------
-------------------------------------------------------------------------------
 Poliana | {'Clean Code', 'The Love', 'The Shack'} | {'email':
'poliana@email.com', 'facebook': 'polianafacebook', 'phone': '+1 55
486848635'}
// to repleace the whole map
UPDATE library.reader SET contacts= {'email': 'just@email.com'} WHERE name
= 'Poliana';
cqlsh> SELECT * from library.reader where name = 'Poliana';
 name    | books                                        | contacts
---------+----------------------------------------------+----------------------
-------
 Poliana | {'Clean Code', 'The Love', 'The Shack'} | {'email':
'just@email.com'}
```

Tupla

Uma tupla é uma combinação de chave e valor. Para o mundo Java, é semelhante ao `java.util.Map.Entry` . Sua estrutura de criação é:

```
tuple_type    ::=  TUPLE '<' cql_type ( ',' cql_type )* '>'
tuple_literal ::=  '(' term ( ',' term )* ')'
```

Para esse exemplo, vamos utilizar a mesma família de coluna de leitores, porém, assumiremos que apenas uma única informação de contato, independente de qual seja, é suficiente. Assim:

```
DROP TABLE IF EXISTS library.reader;

CREATE TABLE IF NOT EXISTS library.reader (
    name text,
    books set<text>,
    contact tuple<text,text>,
    PRIMARY KEY (name)
);
INSERT INTO library.reader (name, books, contact) VALUES ('Poliana', {'The
Shack','The Love','Clean Code'}, ('email', 'poliana@email.com'));
cqlsh> SELECT * FROM library.reader;

 name    | books                                        | contact
---------+----------------------------------------------+----------------------
----------
 Poliana | {'Clean Code', 'The Love', 'The Shack'} | ('email',
'poliana@email.com')
```

> Diferente das coleções, não é possível substituir um único elemento da tupla, como apenas a chave ou o valor. É necessário atualizar todo o campo.

User-Defined Types

O User-Defined Types, ou apenas UDT, é um tipo de dados criado pelo usuário. Esse tipo é criado a partir de um keyspace e segue o mesmo princípio de uma família de colunas, ou seja, será possível criar, alterar e dropar um UDT.

```
create_type_statement ::=  CREATE TYPE [ IF NOT EXISTS ] udt_name
                               '(' field_definition ( ',' field_definition
)* ')'
field_definition      ::=  identifier cql_type
```

Assim como o tipo nativo, para ser utilizado, é necessário defini-lo dentro de uma família de coluna. Por exemplo, para informação de um usuário da biblioteca são necessários o primeiro e o último nome.

```
CREATE TYPE IF NOT EXISTS library.name (
    first_name text,
    last_name text,
);
CREATE TABLE IF NOT EXISTS library.user (
    id text,
    name name,
    PRIMARY KEY (id)
);
INSERT INTO library.user (id, name) values ('otaviojava', {first_name:
'Otavio', last_name: 'Santana'});
INSERT INTO library.user (id, name) values ('poliana', {first_name:
'Poliana', last_name: 'Santana'});
cqlsh> SELECT * FROM library.user;

 id         | name
------------+-----------------------------------------------
    poliana | {first_name: 'Poliana', last_name: 'Santana'}
 otaviojava |  {first_name: 'Otavio', last_name: 'Santana'}
```

Também é possível alterar um UDT. Por exemplo, dado o tipo nome, será adicionado o campo nome do meio.

```
ALTER TYPE library.name ADD middle_name text;
cqlsh> SELECT * FROM library.user;

 id         | name
------------+-----------------------------------------------------------
-----
    poliana | {first_name: 'Poliana', last_name: 'Santana', middle_name:
null}
 otaviojava |  {first_name: 'Otavio', last_name: 'Santana', middle_name:
null}

INSERT INTO library.user (id, name) values ('otaviojava', {first_name:
'Otavio', last_name: 'Santana', middle_name: 'Gonçalves'});
INSERT INTO library.user (id, name) values ('poliana', {first_name:
'Poliana', last_name: 'Santana', middle_name: 'Santos'});

cqlsh> SELECT * FROM library.user;
 id         | name
```

```
------------+-------------------------------------------------------------
-----------
    poliana |   {first_name: 'Poliana', last_name: 'Santana', middle_name:
'Santos'}
 otaviojava | {first_name: 'Otavio', last_name: 'Santana', middle_name:
'Gonçalves'}
```

Também é possível remover remover o UDT como `DROP UDT` .

```
drop_type_statement ::=  DROP TYPE [ IF EXISTS ] udt_name
```

Por exemplo, ao remover o campo nome da família de colunas:

```
ALTER COLUMNFAMILY library.user DROP name;
cqlsh> SELECT * FROM library.user;
 id
------------
    poliana
 otaviojava

DROP TYPE IF EXISTS library.name;
```

As coleções do Cassandra também têm suporte aos UDTs, para isso é necessário utilizar o keyword `frozen` . Vale salientar que o UDT tem o seu funcionamento semelhante ao de uma tupla, ou seja, não é possível alterar um único elemento.

```
DROP TABLE IF EXISTS library.user;
DROP TABLE IF EXISTS library.user;
CREATE TYPE IF NOT EXISTS library.phone (
    country_code int,
    number text,
);
CREATE TABLE IF NOT EXISTS library.user (
    id text,
    phones set<frozen<phone>>,
    PRIMARY KEY (id)
);
```

4.4 Manipulando informação

Nesta seção abordaremos a manipulação de dados, será possível utilizar o famoso CRUD: criar, recuperar, atualizar, deletar as informações.

SELECT

O `SELECT` é o tipo de comando utilizado para recuperar as informações dos dados.

```
select_statement ::=  SELECT [ JSON | DISTINCT ] ( select_clause | '*' )
                      FROM table_name
                      [ WHERE where_clause ]
                      [ GROUP BY group_by_clause ]
                      [ ORDER BY ordering_clause ]
                      [ PER PARTITION LIMIT (integer | bind_marker) ]
                      [ LIMIT (integer | bind_marker) ]
                      [ ALLOW FILTERING ]
select_clause    ::=  selector [ AS identifier ] ( ',' selector [ AS
identifier ] )
selector         ::=  column_name
                      | term
                      | CAST '(' selector AS cql_type ')'
                      | function_name '(' [ selector ( ',' selector )* ]
')'
                      | COUNT '(' '*' ')'
where_clause     ::=  relation ( AND relation )*
relation         ::=  column_name operator term
                      '(' column_name ( ',' column_name )* ')' operator
tuple_literal
                      TOKEN '(' column_name ( ',' column_name )* ')'
operator term
operator         ::=  '=' | '<' | '>' | '<=' | '>=' | '!=' | IN | CONTAINS
| CONTAINS KEY
group_by_clause  ::=  column_name ( ',' column_name )*
ordering_clause  ::=  column_name [ ASC | DESC ] ( ',' column_name [ ASC |
DESC ] )*
```

Por exemplo, na livraria será adicionada uma família de colunas do tipo revista que terá a lista de artigos e o ano do post.

```
DROP COLUMNFAMILY IF EXISTS library.magazine;
CREATE COLUMNFAMILY IF NOT EXISTS library.magazine (
    id text,
    posted_at timestamp,
    articles set<text>,
    pages int,
    PRIMARY KEY (id, posted_at)
);

INSERT INTO library.magazine (id, posted_at, articles, pages) values
('Java Magazine', '2018-01-01',{'Jakarta EE', 'Java 8', 'Cassandra'},
140);
INSERT INTO library.magazine (id, posted_at, articles, pages) values
('Java Magazine', '2017-01-01',{'Java EE 8', 'Java 7', 'NoSQL'}, 100);
```

Dentro de uma query é possível recuperar tanto todos os campos quanto colunas específicas utilizando o mesmo padrão da linguagem SQL.

```
cqlsh> SELECT * FROM library.magazine;

 id            | posted_at                       | articles
| pages
---------------+---------------------------------+------------------------
---------------+-------
 Java Magazine | 2017-01-01 00:00:00.000000+0000 |      {'Java 7', 'Java
EE 8', 'NoSQL'} |   100
 Java Magazine | 2018-01-01 00:00:00.000000+0000 | {'Cassandra', 'Jakarta
EE', 'Java 8'} |   140


cqlsh> SELECT id, posted_at FROM library.magazine;

 id            | posted_at
---------------+---------------------------------
 Java Magazine | 2017-01-01 00:00:00.000000+0000
 Java Magazine | 2018-01-01 00:00:00.000000+0000
```

```
cqlsh> SELECT count(*) FROM library.magazine;

 count
-------
     2
```

Dentro do Cassandra também existem algumas queries de agregação de valores, com a keyword `GROUP BY` , porém, para utilizar esse recurso é necessário que o campo alvo seja uma chave primária.

```
cqlsh> SELECT id, max(pages) FROM library.magazine GROUP BY id;

 id             | system.max(pages)
----------------+-------------------
 Java Magazine |               140

cqlsh> SELECT id, min(pages) FROM library.magazine GROUP BY id;

  id             | system.min(pages)
 ----------------+-------------------
  Java Magazine |               100


cqlsh> SELECT id, sum(pages) FROM library.magazine GROUP BY id;

 id             | system.sum(pages)
----------------+-------------------
 Java Magazine |               240
```

No mundo ideal, todas as queries são realizas a partir da chave, ou seja, sempre haverá o retorno de um único resultado. Porém, em alguns casos, é necessário realizar buscas utilizando índices secundários, trazendo um grande volume de resultados. Uma maneira de evitar essa quantidade é limitando o valor máximo do retorno e o Cassandra tem esse recurso. Para isso, basta utilizar o `LIMIT` .

```
cqlsh> SELECT * FROM library.magazine LIMIT 1;
```

```
 id            | posted_at                       | articles
| pages
---------------+---------------------------------+-----------------------
----------+-------
 Java Magazine | 2017-01-01 00:00:00.000000+0000 | {'Java 7', 'Java EE 8',
'NoSQL'} |   100
```

A ordenação dos dados é feita a partir do `ORDER BY` , porém, ele não é tão poderoso quanto nos bancos relacionais uma vez que a ordenação só é realizada pelos campos do tipo chave.

```
 SELECT * FROM library.magazine where id = 'Java Magazine' ORDER BY
posted_at DESC;
 id            | posted_at                       | articles
| pages
---------------+---------------------------------+-----------------------
---------------+-------
 Java Magazine | 2018-01-01 00:00:00.000000+0000 | {'Cassandra', 'Jakarta
EE', 'Java 8'} |   140
 Java Magazine | 2017-01-01 00:00:00.000000+0000 |      {'Java 7', 'Java
EE 8', 'NoSQL'} |   100


 SELECT * FROM library.magazine where id = 'Java Magazine' ORDER BY
posted_at ASC;

  id            | posted_at                       | articles
| pages
 ---------------+---------------------------------+-----------------------
----------------+-------
  Java Magazine | 2017-01-01 00:00:00.000000+0000 |      {'Java 7', 'Java
EE 8', 'NoSQL'} |   100
  Java Magazine | 2018-01-01 00:00:00.000000+0000 | {'Cassandra', 'Jakarta
EE', 'Java 8'} |   140
```

Em uma query de busca também é possível recuperar os valores no formato *JSON*.

```
cqlsh> SELECT JSON * FROM library.magazine;

 [json]
```

```
------------------------------------------------------------------------
------------------------------------------------------
      {"id": "Java Magazine", "posted_at": "2017-01-01 00:00:00.000Z",
"articles": ["Java 7", "Java EE 8", "NoSQL"], "pages": 100}
 {"id": "Java Magazine", "posted_at": "2018-01-01 00:00:00.000Z",
"articles": ["Cassandra", "Jakarta EE", "Java 8"], "pages": 140}

cqlsh> SELECT JSON id, pages FROM library.magazine;

 [json]
--------------------------------------
 {"id": "Java Magazine", "pages": 100}
 {"id": "Java Magazine", "pages": 140}
```

> É importante salientar que operações de buscas dentro do Cassandra são bem limitadas, de modo que a recomendação é fazer a partir das chaves, deixando o índice em último caso. Existe também o recurso do uso do `ALLOW FILTERING`, porém, utilizar esse recurso demanda um altíssimo impacto de performance.

INSERT

Dentro da cláusula `INSERT` é possível inserir uma linha dentro da família de coluna.

```
insert_statement ::=  INSERT INTO table_name ( names_values | json_clause
)
                      [ IF NOT EXISTS ]
                      [ USING update_parameter ( AND update_parameter )* ]
names_values     ::=  names VALUES tuple_literal
json_clause      ::=  JSON string [ DEFAULT ( NULL | UNSET ) ]
names            ::=  '(' column_name ( ',' column_name )* ')'
```

Para inserir os dados utilizando a cláusula `INSERT` é possível utilizá-la tanto de uma forma bem semelhante ao SQL e quanto como JSON. Por exemplo, utilizando a família de coluna `magazine` anterior.

```
INSERT INTO library.magazine (id, posted_at, articles, pages) values
('Java Magazine', '2017-01-01',{'Java EE 8', 'Java 7', 'NoSQL'}, 100);
INSERT INTO library.magazine JSON '{"id": "Java Magazine", "posted_at":
"2017-01-01", "pages": 10, "articles": ["Java EE 8", "Java 7", "NoSQL"]}';
```

A chave é obrigatória e é a partir dela que é definido onde a informação será enviada através do cluster.

```
INSERT INTO library.magazine JSON '{"posted_at": "2017-01-01", "pages":
10, "articles": ["Java EE 8", "Java 7", "NoSQL"]}';
InvalidRequest: Error from server: code=2200 [Invalid query]
message="Invalid null value in condition for column id"
INSERT INTO library.magazine JSON '{"id": "Java Magazine", "posted_at":
"2017-01-01", "pages": 10}';
INSERT INTO library.magazine JSON '{"id": "Java Magazine", "posted_at":
"2017-01-01"}';
```

Além da verificação da chave primária, não existe nenhuma outra validação, ou seja, é possível diferentes valores com a mesma chave sem nenhum problema. Basicamente, se a chave não existir ela será criada, do contrário, ela será sobrescrita.

```
INSERT INTO library.magazine JSON '{"id": "Java Magazine", "posted_at":
"2017-01-01", "pages": 112}';
INSERT INTO library.magazine JSON '{"id": "Java Magazine", "posted_at":
"2017-01-01", "pages": 121}';
INSERT INTO library.magazine JSON '{"id": "Java Magazine", "posted_at":
"2017-01-01", "pages": 90}';
```

Também é possível definir uma inserção com TTL, que significa o tempo de vida de uma linha em segundos.

```
TRUNCATE library.magazine ;
INSERT INTO library.magazine (id, posted_at) values ('Java Magazine',
'2017-01-01') USING TTL 10;
cqlsh> SELECT * FROM library.magazine WHERE id = 'Java Magazine';

 id             | posted_at                       | articles | pages
----------------+---------------------------------+----------+-------
 Java Magazine | 2017-01-01 00:00:00.000000+0000 |     null |  null
//wait 10 seconds
```

```
cqlsh> SELECT * FROM library.magazine WHERE id = 'Java Magazine';
 id | posted_at | articles | pages
----+-----------+----------+-------

(0 rows)
```

UPDATE

Para atualizar, existe a cláusula `UPDATE`. De uma maneira geral, para o Cassandra, nada mais é que um `INSERT` com o `WHERE`. Assim como no `INSERT`, caso a informação exista, será atualizada, do contrário, será sobrescrita. Por exemplo, dada uma simples família de colunas de bibliotecário, sua criação utilizando a cláusula `UPDATE` ficaria assim:

Criação da família de coluna bibliotecário, como é possível ver é inserido um bibliotecário mesmo utilizando a cláusula `UPDATE`.

```
DROP COLUMNFAMILY IF EXISTS library.librarian;

CREATE COLUMNFAMILY IF NOT EXISTS library.librarian (
    id text,
    name text,
    PRIMARY KEY (id)
);

UPDATE library.librarian set name = 'Ivar' where id = 'ivar';
cqlsh> SELECT * FROM library.librarian;

 id   | name
------+------
 ivar | Ivar

(1 rows)
```

Também é possível atualizar o TTL do registro.

```
UPDATE library.librarian USING TTL 12 set name = 'Daniel' where id =
'daniel';
cqlsh> SELECT * FROM library.librarian where id = 'daniel';
```

```
 id     | name
--------+--------
 daniel | Daniel
//wait 10 seconds
cqlsh> SELECT * FROM library.librarian where id = 'daniel';

 id | name
----+------
```

DELETE

Com a cláusula `DELETE` se pode remover registros dentro do Cassandra, seja apenas uma coluna ou todo o registro, sendo que para qualquer operação é necessária a chave primária, a partition key.

```
DROP COLUMNFAMILY IF EXISTS library.librarian;

CREATE COLUMNFAMILY IF NOT EXISTS library.librarian (
    id text,
    name text,
    PRIMARY KEY (id)
);

INSERT INTO library.librarian JSON '{"name": "Ivar", "id": "ivar"}';
INSERT INTO library.librarian JSON '{"name": "Daniel Dias", "id":
"dani"}';
INSERT INTO library.librarian JSON '{"name": "Gabriela Santana", "id":
"gabriela"}';

cqlsh> SELECT * FROM library.librarian;

 id       | name
----------+------------------
     dani |      Daniel Dias
 gabriela | Gabriela Santana
     ivar |             Ivar

(3 rows)
```

```
DELETE FROM library.librarian where id = 'dani';
cqlsh> SELECT * FROM library.librarian;
 id       | name
----------+------------------
 gabriela | Gabriela Santana
     ivar |             Ivar

(2 rows)
DELETE name FROM library.librarian where id = 'ivar';
cqlsh> SELECT * FROM library.librarian;
 id       | name
----------+------------------
 gabriela | Gabriela Santana
     ivar |             null

(2 rows)
```

BATCH

O `BATCH` permite realizar múltiplas operações de alteração (INSERT, UPDATE e DELETE). As operações dentro de um `BATCH` têm como objetivo realizar diversas operações de maneira atômica, ou seja, ou a operação acontece ou não.

É importante salientar que:

- Os `BATCH` podem conter apenas INSERT, UPDATE e DELETE;
- Os comandos `BATCH` não têm total suporte à transação como os bancos relacionais;
- Todas as operações pertencem a *partition key* para garantir isolamento;
- Por padrão as operações dentro do `BATCH` serão atômicas, assim as operações serão eventualmente completas ou nenhuma operação acontecerá.
- Existe um grande *trade-off* no uso do `BATCH` : de um lado, pode economizar rede na comunicação entre nó coordenador e os outros nós para as operações, do outro lado, pode sobreutilizar um nó para diversas operações, daí a importância de saber usar esse recurso com parcimônia.

```
batch_statement        ::=  BEGIN [ UNLOGGED | COUNTER ] BATCH
                            [ USING update_parameter ( AND
update_parameter )* ]
                            modification_statement ( ';'
modification_statement )*
                            APPLY BATCH
modification_statement ::=  insert_statement | update_statement |
delete_statement
```

Por exemplo, para realizar operações dentro da família de coluna dos bibliotecários.

```
BEGIN BATCH
INSERT INTO library.librarian JSON '{"name": "Ivar", "id": "ivar"}';
INSERT INTO library.librarian JSON '{"name": "Daniel Dias", "id":
"dani"}';
INSERT INTO library.librarian JSON '{"name": "Gabriela Santana", "id":
"gabriela"}';
DELETE FROM library.librarian where id = 'dani';
APPLY BATCH;

cqlsh> SELECT * FROM library.librarian;

 id       | name
----------+------------------
 gabriela | Gabriela Santana
     ivar |             Ivar

(2 rows)
```

View materializada

A desnormalização é a melhor amiga dos bancos de dados do tipo não relacionais, e com o Cassandra isso não é uma exceção. Um dos recursos que podem facilitar nessa modelagem é a criação de views materializadas. Isso serve como um aliado para a desnormalização, ao mesmo tempo que garante a consistência de uma família de coluna. Um ponto importante é que a view materializada não pode sofrer nenhuma alteração.

```
create_materialized_view_statement ::=  CREATE MATERIALIZED VIEW [ IF NOT
EXISTS ] view_name AS
                                            select_statement
                                            PRIMARY KEY '(' primary_key
')'
                                            WITH table_options
```

Por exemplo, considerando novamente o caso do livro e que o ISBN (International Standard Book Number) é o código que os funcionários utilizam para emprestar um livro dentro da biblioteca, seria possível criar a família de coluna da seguinte forma:

```
DROP COLUMNFAMILY IF EXISTS library.book;
CREATE COLUMNFAMILY IF NOT EXISTS library.book (
    isbn bigint,
    name text,
    author text,
    PRIMARY KEY (isbn)
);

INSERT INTO library.book JSON '{"isbn": 1, "name": "Clean Code", "author":
"Robert Cecil Martin"}';
INSERT INTO library.book JSON '{"isbn": 2, "name": "Effective Java",
"author": "Joshua Bloch"}';
INSERT INTO library.book JSON '{"isbn": 3, "name": "The Pragmatic
Programmer", "author": "Andy Hunt"}';
```

Agora é possível realizar buscas de maneira tranquila a partir do ISBN. Porém, considerando que o ISBN é feito de maneira incremental, gostaríamos de retornar os livros mais recentes da biblioteca e, para isso, será criada uma view materializada para ISBN maiores que 3. Assim:

```
CREATE MATERIALIZED VIEW IF NOT EXISTS library.recent_book AS
    SELECT * FROM library.book
    WHERE isbn > 3 AND name IS NOT NULL
    PRIMARY KEY (isbn, name);

Warnings : Materialized views are experimental and are not recommended for
production use.
```

Com a view materializada criada é possível realizar buscas de maneira semelhante ao que se faz a família de colunas.

```
cqlsh> SELECT * FROM library.recent_book;

 isbn | name | author
------+------+--------

INSERT INTO library.book JSON '{"isbn": 4, "name": "Java EE 8 Cookbook",
"author": "Elder Moraes"}';
INSERT INTO library.book JSON '{"isbn": 5, "name": "Best Developer Job
Ever!", "author": "Bruno Souza"}';

cqlsh> SELECT * FROM library.recent_book;
 isbn | name                       | author
------+----------------------------+--------------
    4 |       Java EE 8 Cookbook | Elder Moraes
    5 | Best Developer Job Ever! |  Bruno Souza
```

Também é possível realizar a alteração ou remover a view materializada.

```
DROP MATERIALIZED VIEW library.recent_book;
```

> Até o momento o recurso de view materializada se encontra em caráter experimental.

4.5 Segurança

O Cassandra tem um importante suporte ao recurso de segurança. Assim, é possível definir permissão, criar usuário, criar regras, remover permissão dentre outros. O primeiro passo para aplicarmos isso no projeto é habilitar o recurso de segurança pelo Cassandra. Para isso, é necessário realizar uma mudança dentro do `cassandra.yaml` dentro da pasta `conf` : trocar a linha `authenticator:`

`AllowAllAuthenticator` para `authenticator: PasswordAuthenticator`. Para habilitar o gerenciamento de permissão do Cassandra também é necessário alterar a linha `authorizer: AllowAllAuthorizer` para `authorizer: CassandraAuthorizer`.

Caso esteja utilizando Docker, a solução seria mapear o local de configuração. Assim:

```
docker run --name some-cassandra -p 9042:9042 -v
/my/own/datadir:/var/lib/cassandra -v
/path/to/config/cassandra.yaml:/etc/cassandra/cassandra.yaml -d cassandra
//sample
docker run --name some-cassandra -p 9042:9042 -v
/home/otaviojava/data:/var/lib/cassandra -v
/home/otaviojava/config:/etc/cassandra -d cassandra
```

Feita a habilitação, quando acessar Cassandra novamente haverá uma mensagem de erro:

```
./cqlsh
Connection error: ('Unable to connect to any servers', {'127.0.0.1':
AuthenticationFailed('Remote end requires authentication.',)})
```

Esse erro acontece porque é necessário ter autorização de acesso.

> O superusuário padrão é `cassandra` e a senha `cassandra`.

```
./cqlsh -u cassandra -p cassandra
Connected to Test Cluster at 127.0.0.1:9042.
[cqlsh 5.0.1 | Cassandra 3.11.3 | CQL spec 3.4.4 | Native protocol v4]
Use HELP for help.
cassandra@cqlsh>
```

Para Docker, o acesso é semelhante, basta procurar o ID do contêiner e depois adicionar os parâmetros de usuário e senha.

```
docker exec -it 66e5f38a3815 cqlsh -u cassandra -p cassandra
```

Toda a criação foi definida a partir dos `ROLES`. Por exemplo, é possível criar alguns usuários.

As usuárias `ada` e `alice` criadas:

```
CREATE ROLE ada;
CREATE ROLE alice WITH PASSWORD = 'alice' AND LOGIN = true;
cqlsh> LIST ROLES;
 role      | super | login | options
-----------+-------+-------+---------
       ada | False | False |        {}
     alice | False |  True |        {}
 cassandr
```

Agora é possível, realizar o login com a usuária `alice` .

```
./cqlsh -u alice -p alice
Connected to Test Cluster at 127.0.0.1:9042.
[cqlsh 5.0.1 | Cassandra 3.11.3 | CQL spec 3.4.4 | Native protocol v4]
Use HELP for help.
alice@cqlsh>
```

Uma vez que `alice` não tem permissão de superusuário, não será possível remover campos sensíveis do sistema.

```
alice@cqlsh> DROP KEYSPACE IF EXISTS system_auth;
Unauthorized: Error from server: code=2100 [Unauthorized] message="Cannot
DROP <keyspace system_auth>"
```

É possível alterar a informação de usuário, por exemplo, para fazer com que `alice` tenha permissão de superusuário.

```
ALTER ROLE alice WITH SUPERUSER = true;

cassandra@cqlsh> LIST ROLES;
 role      | super | login | options
-----------+-------+-------+---------
       ada | False | False |        {}
     alice |  True |  True |        {}
 cassandra |  True |  True |        {}
```

Assim como remover usuário:

```
DROP ROLE ada;
cassandra@cqlsh> LIST ROLES;
```

```
 role      | super | login | options
-----------+-------+-------+---------
     alice |  True |  True |        {}
 cassandra |  True |  True |        {}

(2 rows)
```

Com os usuários criados, agora é possível, por exemplo, criar regras e definir as permissões para cada estrutura dentro do banco de dados. Veja que as opções possuem as mesmas estruturas de um banco de dados relacional. As opções disponíveis são:

- CREATE
- ALTER
- DROP
- SELECT
- MODIFY
- AUTHORIZE
- DESCRIBE
- EXECUTE

```
grant_permission_statement ::=  GRANT permissions ON resource TO role_name
permissions                ::=  ALL [ PERMISSIONS ] | permission [
PERMISSION ]
permission                 ::=  CREATE | ALTER | DROP | SELECT | MODIFY |
AUTHORIZE | DESCRIBE | EXECUTE
resource                   ::=  ALL KEYSPACES
                                | KEYSPACE keyspace_name
                                | [ TABLE ] table_name
                                | ALL ROLES
                                | ROLE role_name
                                | ALL FUNCTIONS [ IN KEYSPACE keyspace_name
]
                                | FUNCTION function_name '(' [ cql_type (
',' cql_type )* ] ')'
                                | ALL MBEANS
                                | ( MBEAN | MBEANS ) string
```

Para elucidar o contexto de segurança dentro do Cassandra, imagine o seguinte cenário dentro do sistema de livraria:

- A regra de `user` apenas pode ler dentro do keyspace da biblioteca;
- A regra de `librarian` pode realizar alterações no keyspace da biblioteca;
- A regra de `manager` pode criar tabelas dentro do keyspace da biblioteca.

Com base nessas regras serão criados:

- `ada` como usuária;
- `mike` como bibliotecário;
- `jonh` como manager.

```
CREATE KEYSPACE IF NOT EXISTS library  WITH replication = {'class':
'SimpleStrategy', 'replication_factor' : 3};
//create the role user
CREATE ROLE IF NOT EXISTS user;
GRANT SELECT ON KEYSPACE library TO user;
//create user ada
CREATE ROLE ada WITH PASSWORD = 'ada' AND LOGIN = true;
GRANT user TO ada;
//create the role librarian
CREATE ROLE IF NOT EXISTS librarian;
GRANT MODIFY ON KEYSPACE library TO librarian;
GRANT SELECT ON KEYSPACE library TO librarian;
//create mike
CREATE ROLE mike WITH PASSWORD = 'mike' AND LOGIN = true;
GRANT librarian TO mike;
//create manager
CREATE ROLE IF NOT EXISTS manager;
GRANT ALTER ON KEYSPACE library TO manager;
GRANT CREATE ON KEYSPACE library TO manager;
//create jonh
CREATE ROLE jonh WITH PASSWORD = 'jonh' AND LOGIN = true;
GRANT manager TO jonh;
```

Com todos os usuários criados, o primeiro passo é explorar um pouco da segurança dentro do Cassandra.

```
./cqlsh -u ada -p ada

CREATE COLUMNFAMILY IF NOT EXISTS library.magazine (
    id text,
    pages int,
    PRIMARY KEY (id)
);

Unauthorized: Error from server: code=2100 [Unauthorized] message="User
ada has no CREATE permission on <keyspace library> or any of its parents"

./cqlsh -u jonh -p jonh

CREATE COLUMNFAMILY IF NOT EXISTS library.magazine (
   id text,
    pages int,
    PRIMARY KEY (id)
);

 ./cqlsh -u mike -p mike

INSERT INTO library.magazine JSON '{"id": "new magazine", "pages": 10}';
SELECT * FROM library.magazine;

 id           | pages
--------------+-------
 new magazine |    10

(1 rows)

./cqlsh -u ada -p ada

SELECT * FROM library.magazine ;

 id           | pages
--------------+-------
 new magazine |    10

INSERT INTO library.magazine JSON '{"id": "new magazine", "pages": 10}';
Unauthorized: Error from server: code=2100 [Unauthorized] message="User
```

```
ada has no MODIFY permission on <table library.magazine> or any of its
parents"
```

Com a segurança ativada dentro do Cassandra, os usuários terão permissão para realizar algumas operações a partir do `role`. Essa é uma maneira bem interessante de garantir que apenas sistemas ou pessoas tenham acesso de leitura ou escrita em pontos de que elas realmente precisam.

Conclusão

Neste capítulo foi possível mostrar que o Cassandra Query Language ou CQL tem muitas mais semelhanças com o SQL além do nome, de modo que alguém que já conhece bem os bancos de dados relacionais e sua sintaxe de comunicação terá uma baixa curva de aprendizagem ao aprender o Cassandra.

Um dos pontos importantes que este capítulo explorou foi o uso de segurança que existe dentro do Cassandra. É sempre importante se preocupar com a segurança seja com firewalls de acesso seja com permissão de usuário e senha para pontos do banco de dados.

Capítulo 5
Modelando sua aplicação com Cassandra

Dentro da Ciência de Dados, a modelagem certamente é um dos pontos mais enigmáticos e mais desafiadores. Um erro nesse ponto significará impacto de performance tanto na leitura quanto na escrita de informações no banco de dados. No mundo relacional, quem desenvolve já está acostumado com o conceito de normalização - que é um conjunto de regras que visa, principalmente, à organização dentro do banco de dados relacional para evitar a redundância de dados.

O ponto a ser discutido é que, quando surgiram os bancos relacionais, eles apresentavam o foco de evitar a redundância, afinal, era um grande desafio, já que o preço de armazenamento era algo realmente muito caro. Por exemplo, em 1980 um servidor que armazenava vinte e seis megabytes tinha um custo de quase cinco mil dólares, atualmente, um terabyte custa menos de cinquenta dólares.

Figura 5.1: Um HD com 5MB em 1956 fabricado pela IBM. Fonte: https://thenextweb.com/shareables/2011/12/26/this-is-what-a-5mb-hard-drive-looked-like-is-1956-required-a-forklift/

A consequência da normalização é que, com uma grande complexidade dos dados e a variedade, para manter os dados não duplicados é necessário realizar uma alta quantidade de joins, a união de várias tabelas, e por consequência há um aumento no tempo de resposta dessas queries. O desafio atual se encontra no tempo de resposta e não mais no armazenamento, como era anteriormente.

O objetivo deste capítulo é demonstrar dicas e motivações para realizar uma modelagem dentro do Cassandra.

5.1 Dicas de modelagem

Diferente dos bancos relacionais em que existem as regras de normalização, dentro do Cassandra os princípios de modelagem são definidos muito mais pelo contexto da aplicação ou volumetria. Ou seja, um mesmo sistema de e-commerce, por exemplo, pode ser modelado de maneira diferente a depender dos recursos e volumetria de cada um desses negócios. Como a maioria dos desenvolvedores começa com relacional, as primeiras dicas serão justamente entender quais *não* são os objetivos da modelagem com o Cassandra:

- Minimizar o número de leituras: dentro do Cassandra a escrita é relativamente barata e de maneira eficiente, diferente da leitura; assim, priorize a escrita e tente ler as informações pelo identificador único o máximo possível.
- Normalização ou minimizar dados duplicados: desnormalização e duplicação de dados são as melhores amigas para uma modelagem dentro do Cassandra. Facilitar o máximo possível a leitura é sempre a melhor estratégia dentro do Cassandra.
- Emular: não tente emular ou simular de alguma maneira os bancos relacionais dentro do Cassandra. O resultado será semelhante a usar um garfo como faca.

Um bom primeiro passo para começar a modelagem é saber de quais queries a aplicação precisa. Assim, salientando, criar famílias de colunas que satisfaçam aquela requisição apenas em uma query é o grande objetivo. Existem vários casos de modelagem dentro do mundo Cassandra. Para o próximo tópico, o próximo passo será demonstrar alguns exemplos de modelagem.

Casos um para um

Imagine o seguinte cenário: cada livro (ISBN, nome e ano) tem o seu autor (nome e país).

A primeira pergunta para a modelagem: quais queries queremos suportar?

Para esse caso, gostaríamos de buscar os livros a partir do ISBN uma vez que nesse caso não existe motivo para que se exista uma família de coluna para autores.

```
CREATE KEYSPACE IF NOT EXISTS library  WITH replication = {'class':
'SimpleStrategy', 'replication_factor' : 3};

DROP COLUMNFAMILY IF EXISTS library.book;
DROP TYPE IF EXISTS library.author;

CREATE TYPE IF NOT EXISTS library.author (
    name text,
    country text
);

CREATE COLUMNFAMILY IF NOT EXISTS library.book (
    isbn bigint,
    name text,
    year int,
    author author,
    PRIMARY KEY (isbn)
);
```

No primeiro exemplo, houve a criação do tipo autor do qual ele faz parte.

```
INSERT INTO library.book (isbn, name, year, author) values (1,'Clean
Code', 2008, {name: 'Robert Cecil Martin', country: 'USA'});
INSERT INTO library.book (isbn, name, year, author) values (2,'Clean
Architecture', 2017, {name: 'Robert Cecil Martin', country: 'USA'});
INSERT INTO library.book (isbn, name, year, author) values (3,'Agile
Principles, Patterns, and Practices in C#', 2002, {name: 'Robert Cecil
Martin', country: 'USA'});
INSERT INTO library.book (isbn, name, year, author) values (4,'Java EE 8
Cookbook', 2018, {name: 'Elder Moraes', country: 'Brazil'});
INSERT INTO library.book (isbn, name, year, author) values (5,'Effective
Java', 2001, {name: 'Joshua Bloch', country: 'USA'});
INSERT INTO library.book (isbn, name, year, author) values (6,'Java
```

```
Puzzlers: Traps, Pitfalls, and Corner Cases', 2005, {name: 'Joshua Bloch',
country: 'USA'});
```

No cenário é possível ver o impacto na duplicação das informações, no caso dos autores. Porém, a mudança de nome e nacionalidade de um autor tende a ser algo extremamente raro.

Casos um para N

Seguindo com o exemplo anterior, na relação livro-autor, este exemplo ampliará o número de autores, afinal, existem livros que possuem mais de um escritor. Com isso, existirá uma relação de um livro para N autores. Porém, como o objetivo da leitura não mudou, a modelagem não modificará.

```
CREATE KEYSPACE IF NOT EXISTS library  WITH replication = {'class':
'SimpleStrategy', 'replication_factor' : 3};

DROP COLUMNFAMILY IF EXISTS library.book;
DROP TYPE IF EXISTS library.author;

CREATE TYPE IF NOT EXISTS library.author (
    name text,
    country text
);

CREATE COLUMNFAMILY IF NOT EXISTS library.book (
    isbn bigint,
    name text,
    year int,
    authors set<frozen<author>>,
    PRIMARY KEY (isbn)
);
//INSERT
INSERT INTO library.book (isbn, name, year, authors) values (1,'Design
Patterns: Elements of Reusable Object-Oriented Software', 1994,
{{name: 'Erich Gamma', country: 'Switzerland'}, {name: 'John Vlissides',
country: 'USA'}});
INSERT INTO library.book (isbn, name, year, authors) values (2,'The
Pragmatic Programmer', 1999,
```

```
{{name: 'Andy Hunt', country: 'Switzerland'}, {name: 'Dave Thomas',
country: 'England'}});
```

O impacto segue semelhante ao anterior: um alto grau de dados duplicados. Isso será de grande incômodo para quem vier do mundo relacional, porém, é importante salientar: **desnormalização será a melhor amiga da modelagem no Cassandra**.

Casos N para N

Para o último exemplo, o objetivo é criar uma relação N para N. Seguindo o escopo de negócio de uma biblioteca, é possível pensar na relação entre revistas e artigos porque uma revista possui N artigos da mesma forma que um artigo pode estar em N revistas. Independente disso, no Cassandra a pergunta é a mesma: quais queries o banco de dados deseja suportar? Nesse cenário existirão duas:

- Buscar a revista pelo ISSN
- Buscar o artigo pelo título

```
CREATE KEYSPACE IF NOT EXISTS library  WITH replication = {'class':
'SimpleStrategy', 'replication_factor' : 3};

DROP COLUMNFAMILY IF EXISTS library.magazine;
DROP COLUMNFAMILY IF EXISTS library.article;
DROP TYPE IF EXISTS library.author;
DROP TYPE IF EXISTS library.magazine;
DROP TYPE IF EXISTS library.article;

CREATE TYPE IF NOT EXISTS library.author (
    name text,
    country text
);

CREATE TYPE IF NOT EXISTS library.magazine (
    issn bigint,
    name text,
    year int,
    author frozen<author>
```

```
);

CREATE TYPE IF NOT EXISTS library.article (
    name text,
    year int,
    author frozen<author>
);

CREATE COLUMNFAMILY IF NOT EXISTS library.magazine (
   issn bigint,
   name text,
   editor author,
   articles set<frozen<article>>,
   PRIMARY KEY (issn)
);

CREATE COLUMNFAMILY IF NOT EXISTS library.article (
   title text,
   year int,
   author author,
   magazines set<frozen<magazine>>,
   PRIMARY KEY (title, year)
);
//INSERT
INSERT INTO library.magazine (issn, name, editor, articles) values (1,
'Java Magazine', {name: 'Java Editor', country: 'USA'}, {{name: 'Jakarta
EE', year: 2018, author: {name: 'Elder Moraes', country: 'Brazil'}},
{name: 'Cloud and Docker', year: 2018, author: {name: 'Bruno Souza',
country: 'Brazil'}}});
```

Nesse caso, é bastante simples uma vez que ambas as entidades, revistas e artigos, não necessitam realizar alteração.

Conclusão

Com isso foram apresentados os conceitos de modelagem dentro de um banco de dados Cassandra, demonstrando como ele é diferente de uma base de dados relacional. O núcleo de seu conceito é justamente escrever ao máximo, uma vez que tende a ser uma operação barata, para justamente diminuir o número de

operações na leitura. No mundo ideal é possível criar queries e consultas sem o uso de índices, seja ele como secundário ou com o recurso de `ALLOW FILTERING`, porém, existirá um trabalho gigantesco para gerenciar a duplicação de dados que ocorrerá.

Capítulo 6
Realizando integração com Java

Após toda a teoria, instalação, noção de comunicação com CQL e modelagem, finalmente está na hora de juntar tudo isso e colocar dentro de uma aplicação Java. A integração do banco de dados com a linguagem é uma das coisas mais importantes a serem realizadas dentro de uma aplicação e é esse o objetivo deste capítulo.

6.1 Requisitos mínimos para as demonstrações

Para executar todas as demonstrações será necessário ter o Java 8 instalado, além do Maven superior à versão 3.5.3. No momento em que eu escrevo, o Java se encontra na versão 11, porém, poucos frameworks têm suporte para ele, dessa forma, para manter coerência e facilitar o gerenciamento das variáveis de ambiente, será utilizada a última versão do Java 8.

Como o objetivo do livro é falar sobre o Cassandra, consideramos que você tenha noção de Java 8 e Maven, além de ter os dois instalados e devidamente configurados no sistema operacional favorito. Você utilizar qualquer IDE com que se sinta confortável.

Para facilitar a comparação entre as ferramentas de integração com Java será utilizado um exemplo simples. Com o intuito de diminuir a complexidade da aplicação e não desviar o foco, os exemplos serão baseados apenas em Java SE, ou seja, executando puramente o bom e velho `public static void main(String[])`.

O nosso exemplo será baseado no mesmo caso de uma livraria. Teremos uma biblioteca que terá livros e cada livro terá autor e uma categoria. As consultas desejadas são:

- Buscar o livro a partir do ISBN;
- Buscar os livros a partir da respectiva categoria.

Pensando de uma maneira simples, uma modelagem que atenda a todas as queries e distribua as informações seria assim:

```
CREATE KEYSPACE IF NOT EXISTS library  WITH replication = {'class':
'SimpleStrategy', 'replication_factor' : 3};
DROP COLUMNFAMILY IF EXISTS library.book;
DROP COLUMNFAMILY IF EXISTS library.category;
DROP TYPE IF EXISTS library.book;

CREATE TYPE IF NOT EXISTS library.book (
    isbn bigint,
    name text,
    author text,
    categories set<text>
);

CREATE COLUMNFAMILY IF NOT EXISTS library.book (
    isbn bigint,
    name text,
    author text,
    categories set<text>,
    PRIMARY KEY (isbn)
);


CREATE COLUMNFAMILY IF NOT EXISTS library.category (
  name text PRIMARY KEY,
  books set<frozen<book>>
);
```

Com esse código criamos a estrutura inicial, ou seja, a família de coluna `livro` para a seguir integrar o Cassandra com o código Java.

6.2 Esqueleto dos projetos exemplos

Como já foi mencionado, os exemplos serão baseados no Java SE, assim, o nosso primeiro passo com o Java é criar o projeto. Utilizaremos o *maven archetype* para acelerar a criação desse projeto maven. Acesse o terminal e execute o seguinte comando:

```
mvn archetype:generate -DgroupId=com.mycompany.app -DartifactId=my-app -
DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false
```

Um ponto importante é que dentro do `pom.xml` é importante atualizá-lo para ter suporte ao Java 8. Assim, o arquivo ficaria desta forma:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.mycompany.app</groupId>
    <artifactId>my-app</artifactId>
    <packaging>jar</packaging>
    <version>1.0-SNAPSHOT</version>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>
    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <configuration>
                    <source>8</source>
                    <target>8</target>
                </configuration>
            </plugin>
        </plugins>
    </build>
    <name>my-app</name>
    <url>http://maven.apache.org</url>
    <dependencies>
        <dependency>
            <groupId>junit</groupId>
```

```
            <artifactId>junit</artifactId>
            <version>3.8.1</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
</project>
```

> Para cada novo projeto uma boa estratégia seria criá-lo utilizando o mesmo esqueleto.

6.3 Utilizando o driver do Cassandra

Para começar a desenvolver o código, o primeiro passo é adicionar a dependência do driver dentro do arquivo que gerencia as dependências do Maven, ou seja, o `pom.xml` dentro da tag `dependencies`:

```
<dependency>
    <groupId>com.datastax.oss</groupId>
    <artifactId>java-driver-core</artifactId>
    <version>${data.stax.version}</version>
</dependency>
<dependency>
    <groupId>com.datastax.oss</groupId>
    <artifactId>java-driver-query-builder</artifactId>
    <version>${data.stax.version}</version>
</dependency>
```

> Para facilitar, não ativaremos a segurança e estaremos baseados em uma instalação local ou com o Docker com o seguinte comando: `docker run -d --name casandra-instance -p 9042:9042 cassandra .`

Com a dependência dentro do projeto, o próximo passo é iniciar a comunicação. O `CqlSession` é a classe que representa a estrutura de nós do Cassandra. Com ele, é possível utilizar o `try-resource`, dessa maneira, tão logo o código seja encerrado o cluster também se encerra.

```
public class App {

    private static final String KEYSPACE = "library";
    private static final String COLUMN_FAMILY = "book";

    public static void main(String[] args) {
        try (CqlSession session = CqlSession.builder().build()) {

            Map<String, Term> cleanCode = createInsertQuery(new Object[]
{1, "Clean Code", "Robert Cecil Martin",
                    Set.of("Java", "OO")});
            Map<String, Term> cleanArchitecture = createInsertQuery(new
Object[]{2, "Clean Architecture",
                    "Robert Cecil Martin",
                    Set.of("Good practice")});
            Map<String, Term> effectiveJava = createInsertQuery(new
Object[]{3, "Effective Java", "Joshua Bloch",
                    Set.of("Java", "Good practice")});
            Map<String, Term> nosql = createInsertQuery(new Object[]{4,
"Nosql Distilled", "Martin Fowler",
                    Set.of("NoSQL", "Good practice")});

            session.execute(QueryBuilder.insertInto(KEYSPACE,
COLUMN_FAMILY).values(cleanCode).build());
            session.execute(QueryBuilder.insertInto(KEYSPACE,
COLUMN_FAMILY).values(cleanArchitecture).build());
            session.execute(QueryBuilder.insertInto(KEYSPACE,
COLUMN_FAMILY).values(effectiveJava).build());
            session.execute(QueryBuilder.insertInto(KEYSPACE,
COLUMN_FAMILY).values(nosql).build());
            session.execute(QueryBuilder.insertInto(KEYSPACE,
COLUMN_FAMILY).values(cleanCode).build());

            ResultSet resultSet =
```

```
session.execute(QueryBuilder.selectFrom(KEYSPACE,
COLUMN_FAMILY).all().build());
            for (Row row : resultSet) {
                Long isbn = row.getLong("isbn");
                String name = row.getString("name");
                String author = row.getString("author");
                Set<String> categories = row.getSet("categories",
String.class);
                System.out.println(String.format(" the result is %s %s %s
%s", isbn, name, author, categories));
            }
        }

    }

    private static Map<String, Term>  createInsertQuery(Object[]
parameters) {
        return Map.of("isbn", literal(parameters[0]), "name",
literal(parameters[1]),
                "author", literal(parameters[2]),
                "categories", literal(parameters[3]));

    }

}
```

Na primeira classe de interação com o Cassandra, é possível perceber o uso intenso do Cassandra Query Language, que explicamos no capítulo 4. Todas as operações com o CQL são facilitadas a partir da classe `QueryBuilder` , que é uma utilitária que contém diversos métodos que facilitam a vida do desenvolvedor para criar CQL para uma aplicação.

Na nossa primeira interação realizamos uma inserção e uma busca de todos os valores dentro do banco de dados, mas esse tipo de busca não é muito comum e tampouco otimizado. Como mencionamos no capítulo de modelagem, o mais recomendado é sempre realizar uma busca pela chave. Assim, na nossa próxima interação realizaremos buscas a partir da chave. Para facilitar um

pouco o código, criaremos um `Consumer` para logar o resultado (não será nada sofisticado, apenas o bom e velho `System.out.println`).

```
public class App2 {

    private static final String KEYSPACE = "library";
    private static final String COLUMN_FAMILY = "book";

    public static void main(String[] args) {
        try (CqlSession session = CqlSession.builder().build()) {

            Map<String, Term> cleanCode = createInsertQuery(new Object[]
{1, "Clean Code", "Robert Cecil Martin",
                    Set.of("Java", "OO")});
            Map<String, Term> cleanArchitecture = createInsertQuery(new
Object[]{2, "Clean Architecture",
                    "Robert Cecil Martin",
                    Set.of("Good practice")});
            Map<String, Term> effectiveJava = createInsertQuery(new
Object[]{3, "Effective Java", "Joshua Bloch",
                    Set.of("Java", "Good practice")});
            Map<String, Term> nosql = createInsertQuery(new Object[]{4,
"Nosql Distilled", "Martin Fowler",
                    Set.of("NoSQL", "Good practice")});

            session.execute(QueryBuilder.insertInto(KEYSPACE,
COLUMN_FAMILY).values(cleanCode).build());
            session.execute(QueryBuilder.insertInto(KEYSPACE,
COLUMN_FAMILY).values(cleanArchitecture).build());
            session.execute(QueryBuilder.insertInto(KEYSPACE,
COLUMN_FAMILY).values(effectiveJava).build());
            session.execute(QueryBuilder.insertInto(KEYSPACE,
COLUMN_FAMILY).values(nosql).build());
            session.execute(QueryBuilder.insertInto(KEYSPACE,
COLUMN_FAMILY).values(cleanCode).build());

            Consumer<Row> log = row -> {
                Long isbn = row.getLong("isbn");
                String name = row.getString("name");
                String author = row.getString("author");
                Set<String> categories = row.getSet("categories",
```

```
String.class);
                System.out.println(String.format(" the result is %s %s %s
%s", isbn, name, author, categories));
            };

            findById(session,1L, log);

            deleteById(session, 1L);

            PreparedStatement prepare = session.prepare("select * from
library.book where isbn = ?");
            BoundStatement statement = prepare.bind(2L);
            ResultSet resultSet = session.execute(statement);
            resultSet.forEach(log);

        }

    }

    private static void deleteById(CqlSession session, Long isbn) {
        session.execute(QueryBuilder.deleteFrom(KEYSPACE, COLUMN_FAMILY)

.where(Relation.column("isbn").isEqualTo(QueryBuilder.literal(isbn))).buil
d());

    }

    private static void findById(CqlSession session, long isbn,
Consumer<Row> log) {
        ResultSet resultSet =
session.execute(QueryBuilder.selectFrom(KEYSPACE, COLUMN_FAMILY)

.all().where(Relation.column("isbn").isEqualTo(QueryBuilder.literal(isbn))
).build());
        resultSet.forEach(log);
    }

    private static Map<String, Term> createInsertQuery(Object[]
parameters) {
        return Map.of("isbn", literal(parameters[0]), "name",
literal(parameters[1]),
```

```
                "author", literal(parameters[2]),
                "categories", literal(parameters[3]));

    }

}
```

Com o código anterior foi possível realizar as operações básicas do CRUD com os tipos triviais no Cassandra. Seguindo e evoluindo o nosso exemplo, realizaremos manipulações com um tipo customizado do Cassandra, ou UDT dentro da família de coluna `Category` .

```
public class App3 {

    private static final String KEYSPACE = "library";
    private static final String TYPE = "book";
    private static final String COLUMN_FAMILY = "category";

    public static void main(String[] args) {
        try (CqlSession session = CqlSession.builder().build()) {

            UserDefinedType userType =
                    session.getMetadata()
                            .getKeyspace(KEYSPACE)
                            .flatMap(ks -> ks.getUserDefinedType(TYPE))
                            .orElseThrow(() -> new
IllegalArgumentException("Missing UDT definition"));

            UdtValue cleanCode = getValue(userType, 1, "Clean Code",
"Robert Cecil Martin", Set.of("Java", "OO", "Good practice", "Design"));
            UdtValue cleanArchitecture = getValue(userType, 2, "Clean
Architecture", "Robert Cecil Martin", Set.of("OO", "Good practice"));
            UdtValue effectiveJava = getValue(userType, 3, "Effective
Java", "Joshua Bloch", Set.of("Java", "OO", "Good practice"));
            UdtValue nosql = getValue(userType, 4, "Nosql Distilled",
"Martin Fowler", Set.of("NoSQL", "Good practice"));

            session.execute(QueryBuilder.insertInto(KEYSPACE,
COLUMN_FAMILY)
```

```
                    .values(createCondition("Java", Set.of(cleanCode,
effectiveJava))).build());
            session.execute(QueryBuilder.insertInto(KEYSPACE,
COLUMN_FAMILY)
                    .values(createCondition("OO", Set.of(cleanCode,
effectiveJava, cleanArchitecture))).build());
            session.execute(QueryBuilder.insertInto(KEYSPACE,
COLUMN_FAMILY)
                    .values(createCondition("Good practice",
                            Set.of(cleanCode, effectiveJava,
cleanArchitecture, nosql))).build());
            session.execute(QueryBuilder.insertInto(KEYSPACE,
COLUMN_FAMILY)
                    .values(createCondition("NoSQL",
Set.of(nosql))).build());

            ResultSet resultSet =
session.execute(QueryBuilder.selectFrom(KEYSPACE,
COLUMN_FAMILY).all().build());
            for (Row row : resultSet) {
                String name = row.getString("name");
                Set<UdtValue> books = row.getSet("books", UdtValue.class);
                Set<String> logBooks = new HashSet<>();
                for (UdtValue book : books) {
                    long isbn = book.getLong("isbn");
                    String bookName = book.getString("name");
                    String author = book.getString("author");
                    logBooks.add(String.format(" %d %s %s", isbn,
bookName, author));
                }
                System.out.println(String.format("The result %s %s", name,
logBooks));

            }
        }

    }

    private static UdtValue getValue(UserDefinedType userType, long isbn,
String name, String author, Set<String> categories) {
        UdtValue udtValue = userType.newValue();
```

```
        TypeCodec<Object> bigIntCodec =
CodecRegistry.DEFAULT.codecFor(userType.getFieldTypes().get(0));
        TypeCodec<Object> textCodec =
CodecRegistry.DEFAULT.codecFor(userType.getFieldTypes().get(1));
        TypeCodec<Object> setCodec =
CodecRegistry.DEFAULT.codecFor(userType.getFieldTypes().get(3));
        udtValue.set("isbn", isbn, bigIntCodec);
        udtValue.set("name", name, textCodec);
        udtValue.set("author", author, textCodec);
        udtValue.set("categories", categories, setCodec);
        return udtValue;

    }

    private static Map<String, Term> createCondition(String name,
Set<UdtValue> books) {
        return Map.of("name", QueryBuilder.literal(name), "books",
QueryBuilder.literal(books));
    }

}
```

Agora conseguimos realizar a inserção de um tipo customizado do Cassandra o `UDT` , ou seja, além de realizar a inserção também recuperamos um tipo `UDT` dentro do banco de dados Cassandra.

O Driver do Cassandra também possui o conceito de `PreparedStatement` que permite que sejam criadas queries dinamicamente, muito semelhante aos bancos de dados relacionais.

```
public class App4 {

    private static final String KEYSPACE = "library";
    private static final String TYPE = "book";
    private static final String COLUMN_FAMILY = "category";

    public static void main(String[] args) {
        try (CqlSession session = CqlSession.builder().build()) {

            UserDefinedType userType =
```

```
                    session.getMetadata()
                            .getKeyspace(KEYSPACE)
                            .flatMap(ks -> ks.getUserDefinedType(TYPE))
                            .orElseThrow(() -> new
IllegalArgumentException("Missing UDT definition"));

            UdtValue cleanCode = getValue(userType, 1, "Clean Code",
"Robert Cecil Martin", Set.of("Java", "OO", "Good practice", "Design"));
            UdtValue cleanArchitecture = getValue(userType, 2, "Clean
Architecture", "Robert Cecil Martin", Set.of("OO", "Good practice"));
            UdtValue effectiveJava = getValue(userType, 3, "Effective
Java", "Joshua Bloch", Set.of("Java", "OO", "Good practice"));
            UdtValue nosql = getValue(userType, 4, "Nosql Distilled",
"Martin Fowler", Set.of("NoSQL", "Good practice"));

            session.execute(QueryBuilder.insertInto(KEYSPACE,
COLUMN_FAMILY)
                    .values(createCondition("Java", Set.of(cleanCode,
effectiveJava))).build());
            session.execute(QueryBuilder.insertInto(KEYSPACE,
COLUMN_FAMILY)
                    .values(createCondition("OO", Set.of(cleanCode,
effectiveJava, cleanArchitecture))).build());
            session.execute(QueryBuilder.insertInto(KEYSPACE,
COLUMN_FAMILY)
                    .values(createCondition("Good practice",
                            Set.of(cleanCode, effectiveJava,
cleanArchitecture, nosql))).build());
            session.execute(QueryBuilder.insertInto(KEYSPACE,
COLUMN_FAMILY)
                    .values(createCondition("NoSQL",
Set.of(nosql))).build());

            Consumer<Row> log = row -> {
                String name = row.getString("name");
                Set<UdtValue> books = row.getSet("books", UdtValue.class);
                Set<String> logBooks = new HashSet<>();
                for (UdtValue book : books) {
                    long isbn = book.getLong("isbn");
                    String bookName = book.getString("name");
                    String author = book.getString("author");
```

```
                    logBooks.add(String.format(" %d %s %s", isbn,
bookName, author));
                }
                System.out.println(String.format("The result %s %s", name,
logBooks));
            };

            findById(session, "OO", log);
            findById(session, "Good practice", log);
            deleteById(session, "OO");

            PreparedStatement prepare = session.prepare("select * from
library.category where name = ?");
            BoundStatement statement = prepare.bind("Java");
            ResultSet resultSet = session.execute(statement);
            resultSet.forEach(log);
        }

    }

    private static void findById(CqlSession session, String name,
Consumer<Row> log) {
        ResultSet resultSet =
session.execute(QueryBuilder.selectFrom(KEYSPACE, COLUMN_FAMILY)

.all().where(Relation.column("name").isEqualTo(QueryBuilder.literal(name))
).build());
        resultSet.forEach(log);
    }

    private static void deleteById(CqlSession session, String name) {
        session.execute(QueryBuilder.deleteFrom(KEYSPACE, COLUMN_FAMILY)

.where(Relation.column("name").isEqualTo(QueryBuilder.literal(name))).buil
d());

    }

    private static UdtValue getValue(UserDefinedType userType, long isbn,
String name, String author, Set<String> categories) {
        UdtValue udtValue = userType.newValue();
```

```
        TypeCodec<Object> bigIntCodec =
CodecRegistry.DEFAULT.codecFor(userType.getFieldTypes().get(0));
        TypeCodec<Object> textCodec =
CodecRegistry.DEFAULT.codecFor(userType.getFieldTypes().get(1));
        TypeCodec<Object> setCodec =
CodecRegistry.DEFAULT.codecFor(userType.getFieldTypes().get(3));
        udtValue.set("isbn", isbn, bigIntCodec);
        udtValue.set("name", name, textCodec);
        udtValue.set("author", author, textCodec);
        udtValue.set("categories", categories, setCodec);
        return udtValue;
    }

    private static Map<String, Term> createCondition(String name,
Set<UdtValue> books) {
        return Map.of("name", QueryBuilder.literal(name), "books",
QueryBuilder.literal(books));
    }
}
```

Com isso, realizamos um exemplo completo com o Driver do Cassandra, e completamos todo o ciclo do CRUD dentro da família de coluna `Category`. Utilizar o Driver é muito bom uma vez que temos acesso a todas as queries dentro do CQL, porém, isso acaba dando um grande trabalho, afinal, para cada operação temos que fazer a conversão para o tipo Java a todo o momento.

Será que existe alguma maneira de diminuir a quantidade de código para trabalhar com o Cassandra? A resposta é sim, no Cassandra também há ferramentas que atuam para converter as entidades em códigos nativos: os Mappers.

6.4 Utilizando o Mapper

Para o nosso primeiro contato, será utilizado o Mapper que é mantido pela mesma empresa que mantém o Driver. De fato, ele é

uma camada acima da camada do Driver.

A grande vantagem do Mapper é a redução de código duplicado, o que facilita a integração entre as entidades Java e a aplicação.

```
<dependencies>
<dependency>
    <groupId>com.datastax.oss</groupId>
    <artifactId>java-driver-mapper-runtime</artifactId>
    <version>${data.stax.version}</version>
</dependency>
...
<dependencies>

<build>
<plugins>
    <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.8.1</version>
        <configuration>
            <source>11</source>
            <target>11</target>
            <annotationProcessorPaths>
                <path>
                    <groupId>com.datastax.oss</groupId>
                    <artifactId>java-driver-mapper-processor</artifactId>
                    <version>${data.stax.version}</version>
                </path>
            </annotationProcessorPaths>
        </configuration>
    </plugin>
</plugins>
</build>
```

Para o mesmo exemplo que realiza a manipulação da família de coluna `Book`, o primeiro passo é o mapeamento, que nada mais é que uma entidade cujos atributos são anotados. Ao ver o código, é possível perceber que as anotações são bem similares para quem veio do mundo do JPA:

- A anotação `Table` é para indicar que a classe é uma família de coluna.
- `Column` indica que o campo será mapeado para o banco de dados.
- `PartitionKey` indica que aquele atributo desempenha um papel especial dentro da família de coluna, que é a de chave primária.

```
@Entity
public class Book {

    @PartitionKey
    private Long isbn;

    private String name;

    private String author;

    private Set<String> categories;

    //getters and setters
}
```

No primeiro contato com o Mapper é possível notar uma grande redução de código. Um ponto importante para ressaltar é o `BookDao`, que serve para gerenciar as entidades e a comunicação entre CQL e Java.

```
@Dao
public interface BookDao {

    @Select
    Book findById(Long id);

    @Insert
    void save(Book book);

    @Delete
    void delete(Book book);

    @Query("SELECT * FROM library.book")
```

```
    PagingIterable<Book> map();

    @Insert
    CompletableFuture<Book> saveAsync(Book book);

    @Select
    CompletableFuture<Book> findByIdAsync(Long id);
}


@Mapper
public interface InventoryMapper {
    @DaoFactory
    BookDao getBookDao(@DaoKeyspace CqlIdentifier keyspace);

    @DaoFactory
    CategoryDao getCategoryDao(@DaoKeyspace CqlIdentifier keyspace);
}


public class App {

    private static final String KEYSPACE = "library";

    public static void main(String[] args) {
        try (CqlSession session = CqlSession.builder().build()) {

            InventoryMapper inventoryMapper = new
InventoryMapperBuilder(session).build();
            BookDao mapper =
inventoryMapper.getBookDao(CqlIdentifier.fromCql(KEYSPACE));
            Book cleanCode = getBook(1L, "Clean Code", "Robert Cecil
Martin", Set.of("Java", "OO"));
            Book cleanArchitecture = getBook(2L, "Clean Architecture",
"Robert Cecil Martin", Set.of("Good practice"));
            Book effectiveJava = getBook(3L, "Effective Java", "Joshua
Bloch", Set.of("Java", "Good practice"));
            Book nosql = getBook(4L, "Nosql Distilled", "Martin Fowler",
Set.of("NoSQL", "Good practice"));

            mapper.save(cleanCode);
```

```
            mapper.save(cleanArchitecture);
            mapper.save(effectiveJava);
            mapper.save(nosql);

            for (Book book : mapper.map()) {
                System.out.println("The result: " + book);
            }
        }

    }

    private static Book getBook(long isbn, String name, String author,
Set<String> categories) {
        Book book = new Book();
        book.setIsbn(isbn);
        book.setName(name);
        book.setAuthor(author);
        book.setCategories(categories);
        return book;
    }

}

public class App2 {

    private static final String KEYSPACE = "library";

    public static void main(String[] args) {
        try (CqlSession session = CqlSession.builder().build()) {

            InventoryMapper inventoryMapper = new
InventoryMapperBuilder(session).build();
            BookDao mapper =
inventoryMapper.getBookDao(CqlIdentifier.fromCql(KEYSPACE));

            Book cleanCode = getBook(1L, "Clean Code", "Robert Cecil
Martin", Set.of("Java", "OO"));
            Book cleanArchitecture = getBook(2L, "Clean Architecture",
"Robert Cecil Martin", Set.of("Good practice"));
            Book effectiveJava = getBook(3L, "Effective Java", "Joshua
Bloch", Set.of("Java", "Good practice"));
```

```
            Book nosql = getBook(4L, "Nosql Distilled", "Martin Fowler",
Set.of("NoSQL", "Good practice"));

            mapper.save(cleanCode);
            mapper.save(cleanArchitecture);
            mapper.save(effectiveJava);
            mapper.save(nosql);

            Book book = mapper.findById(1L);
            System.out.println("Book found: " + book);

            mapper.delete(book);
            System.out.println("Book found: " + mapper.findById(1L));

        }

    }

    private static Book getBook(long isbn, String name, String author,
Set<String> categories) {
        Book book = new Book();
        book.setIsbn(isbn);
        book.setName(name);
        book.setAuthor(author);
        book.setCategories(categories);
        return book;
    }

}
```

O código cria quatro tipos de livros e os armazena no banco de dados utilizando a API. Em seguida, realiza a busca pela chave e, finalmente, buscas e executando sem se preocupar com a conversão da entidade `Book` .

O próximo passo é evoluir o código adicionando o `UDT` com a família de coluna `Category` .

```
@Entity
public class Category {
```

```
    @PartitionKey
    private String name;

    private Set<BookType> books;

    //getters and setters

}

@Entity
public class BookType {

    private Long isbn;

    private String name;

    private String author;

    private Set<String> categories;

    //getters and setters
}

@Dao
public interface CategoryDao {

    @Select
    Category findById(String id);

    @Insert
    void save(Category book);

    @Delete
    void delete(Category book);

    @Query("SELECT * FROM library.category")
    PagingIterable<Category> map();

}
```

Modelagem criada e pronta para a interação no banco de dados com o tipo `UDT` que será representada pela classe `Category`. Essa classe é bem semelhante à entidade `Book`, que criamos anteriormente, com a diferença na forma como a informação será informada. Um ponto a salientar é que as anotações são bastante intuitivas.

```
public class App3 {

    private static final String KEYSPACE = "library";

    public static void main(String[] args) {
        try (CqlSession session = CqlSession.builder().build()) {

            InventoryMapper inventoryMapper = new
InventoryMapperBuilder(session).build();
            CategoryDao mapper =
inventoryMapper.getCategoryDao(CqlIdentifier.fromCql(KEYSPACE));

            BookType cleanCode = getBook(1L, "Clean Code", "Robert Cecil
Martin", Set.of("Java", "OO"));
            BookType cleanArchitecture = getBook(2L, "Clean Architecture",
"Robert Cecil Martin", Set.of("Good practice"));
            BookType effectiveJava = getBook(3L, "Effective Java", "Joshua
Bloch", Set.of("Java", "Good practice"));
            BookType nosqlDistilled = getBook(4L, "Nosql Distilled",
"Martin Fowler", Set.of("NoSQL", "Good practice"));

            Category java = getCategory("Java", Sets.newHashSet(cleanCode,
effectiveJava));
            Category oo = getCategory("OO", Sets.newHashSet(cleanCode,
effectiveJava, cleanArchitecture));
            Category goodPractice = getCategory("Good practice",
Sets.newHashSet(cleanCode, effectiveJava, cleanArchitecture,
nosqlDistilled));
            Category nosql = getCategory("NoSQL",
Sets.newHashSet(nosqlDistilled));

            mapper.save(java);
            mapper.save(oo);
```

```
            mapper.save(goodPractice);
            mapper.save(nosql);

            PagingIterable<Category> categories = mapper.map();
            StreamSupport.stream(categories.spliterator(),
false).forEach(System.out::println);
        }

    }

    private static Category getCategory(String name, Set<BookType> books)
{
        Category category = new Category();
        category.setName(name);
        category.setBooks(books);
        return category;
    }

    private static BookType getBook(long isbn, String name, String author,
Set<String> categories) {
        BookType book = new BookType();
        book.setIsbn(isbn);
        book.setName(name);
        book.setAuthor(author);
        book.setCategories(categories);
        return book;
    }
}
```

O código é bem semelhante à interação da classe `Book` , a única exceção é o uso da classe `Category.class` como parâmetro dentro do Mapper e, obviamente, serão criadas instâncias do tipo `Category` para a persistência da informação.

> Também é possível utilizar esse tipo de mapeamento integrado com o Quarkus, para saber mais: https://quarkus.io/guides/cassandra

Conclusão

Neste capítulo se obteve o marco da integração entre o Cassandra e um aplicativo Java. Também foi possível ver a diferença de um framework que realiza a comunicação de baixo nível, semelhante ao JDBC, e um Mapper que, ao mesmo tempo que facilita o desenvolvimento, aumenta a responsabilidade para que o desenvolvedor não cometa erros ao esquecer que existe uma quebra de paradigma entre a aplicação e o banco de dados.

Capítulo 7
Criando um aplicativo com Hibernate

No mundo relacional existem diversos frameworks ORM que facilitam a integração entre a aplicação e o objeto relacional, dentre eles, o mais famoso no mundo Java é o Hibernate. Trata-se de um ORM Java open source criado por Gavin King e, atualmente, é desenvolvido pela RedHat.

No mundo Java, existe um processo de padronização, que é a JSR, Java Speficication Request, regido pelo JCP, Java Community Process (no momento que eu escrevo este capítulo existe um processo de migração do processo do JCP para Eclipse Foundation com um novo nome, que será discutido no capítulo sobre Jakarta). As especificações garantem diversos benefícios, dentre eles diversas empresas, instituições acadêmicas e membros da comunidade contribuindo nos projetos, totalmente orientado à comunidade. E o principal: o bloqueio do *vendor lock-in*, assim, o desenvolvedor não tem o risco de o projeto ser descontinuado por uma empresa, uma vez que poderá ser substituída por outras. A especificação entre o mundo Java e o banco relacional é o JPA, que faz parte da plataforma Java EE, agora Jakarta EE.

Assim como o Spring, o Hibernate possui diversos subprojetos para facilitar o mundo de desenvolvimento mais focado no mundo de persistência de dados, por exemplo, Hibernate Search, Hibernate Validator, dentre outras coisas. Neste capítulo, falemos um pouco mais sobre o Hibernate e sua relação com o mundo não relacional com o Hibernate OGM.

7.1 O que é Hibernate OGM?

Com o objetivo de facilitar a integração e diminuir a curva de aprendizagem para aprender bancos de dados não relacionais, nasceu o Hibernate OGM. Sua abordagem é bastante simples: utilizar a API JPA, que os desenvolvedores Java já conhecem, com bancos de dados NoSQL. Atualmente Hibernate OGM tem suporte para diversos projetos:

- Cassandra
- CouchDB
- EhCache
- Apache Ignite
- Redis

Exemplo prático com Hibernate OGM

Com uma breve introdução da história do Hibernate e da importância desse projeto no mundo Java, seguiremos para a parte prática. Sua configuração é definida em duas partes:

1. A inserção de dependências, a partir do Maven.
2. A adição das configurações do banco de dados a partir do arquivo `persistence.xml` dentro da pasta `META-INF` .

```
<dependency>
    <groupId>org.hibernate.ogm</groupId>
    <artifactId>hibernate-ogm-cassandra</artifactId>
    <version>5.1.0.Final</version>
</dependency>
<dependency>
    <groupId>org.jboss.logging</groupId>
    <artifactId>jboss-logging</artifactId>
    <version>3.3.0.Final</version>
</dependency>
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-search-orm</artifactId>
    <version>5.6.1.Final</version>
</dependency>
```

A configuração entre o banco de dados e o framework Java é realizada a partir do arquivo `persistence.xml`, seguindo a linha da configuração do JPA, ou seja, não existe nenhuma novidade para quem já conhece essa especificação.

- `hibernate.ogm.datastore.provider`: define qual implementação o Hibernate utilizará, nesse caso, a implementação que utiliza o Cassandra.
- `hibernate.ogm.datastore.database`: para o Cassandra é o `keyspace`, ou seja, para esse exemplo será o `library`.
- `hibernate.search.default.directory_provider` e `hibernate.search.default.indexBase`: uma das dependências existentes no Hibernate OGM Cassandra é o Hiberante Search, que é a parte do Hibernate que oferece o `full-search` para os objetos gerenciados pelo Hibernate. É possível realizar buscas utilizando o Lucene ou Elasticsearch.

```
<?xml version="1.0" encoding="utf-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
             version="2.0">

    <persistence-unit name="hibernate">
        <provider>org.hibernate.ogm.jpa.HibernateOgmPersistence</provider>
        <class>com.nosqlxp.cassandra.Book</class>
        <properties>
            <property name="hibernate.ogm.datastore.provider"
value="org.hibernate.ogm.datastore.cassandra.impl.CassandraDatastoreProvid
er"/>
            <property name="hibernate.ogm.datastore.host"
value="localhost:9042"/>
            <property name="hibernate.ogm.datastore.create_database"
value="true"/>
            <property name="hibernate.ogm.datastore.database"
value="library"/>
            <property name="hibernate.search.default.directory_provider"
value="filesystem"/>
```

```
            <property name="hibernate.search.default.indexBase"
value="/tmp/lucene/data"/>
        </properties>
    </persistence-unit>
</persistence>
```

Com a parte de infraestrutura pronta dentro do projeto, o próximo passo é a modelagem. Toda anotação do modelo é feita utilizando as anotações do JPA, o que facilita para quem já conhece a API de persistência relacional dentro do mundo Java.

```
@Entity(name = "book")
@Indexed
@Analyzer(impl = StandardAnalyzer.class)
public class Book {

    @Id
    @DocumentId
    private Long isbn;

    @Column
    @Field(analyze = Analyze.NO)
    private String name;

    @Column
    @Field
    private String author;

    @Column
    @Field
    private String category;

   //getter and setter
}
```

Para atender ao requisito de buscar tanto pela chave quanto pela categoria, utilizaremos o recurso de motor de busca com o Apache Lucene através do Hibernate Search. Esse recurso é realmente muito interessante, principalmente, para permitir a busca de campos que não sejam a chave. Porém, isso incrementa a dificuldade do

projeto, uma vez que existirá o desafio para sincronizar as informações dentro do banco de dados e do motor de busca.

```
public class App {


    public static void main(String[] args) {
        EntityManagerFactory managerFactory =
Persistence.createEntityManagerFactory("hibernate");
        EntityManager manager = managerFactory.createEntityManager();
        manager.getTransaction().begin();

        Book cleanCode = getBook(1L, "Clean Code", "Robert Cecil Martin");
        Book cleanArchitecture = getBook(2L, "Clean Architecture", "Robert
Cecil Martin");
        Book agile = getBook(3L, "Agile Principles, Patterns, and
Practices in C#", "Robert Cecil Martin");
        Book effectiveJava = getBook(4L, "Effective Java", "Joshua
Bloch");
        Book javaConcurrency = getBook(5L, "Java Concurrency", "Robert
Cecil Martin");

        manager.merge(cleanCode);
        manager.merge(cleanArchitecture);
        manager.merge(agile);
        manager.merge(effectiveJava);
        manager.merge(javaConcurrency);
        manager.getTransaction().commit();

        Book book = manager.find(Book.class, 1L);
        System.out.println("book: " + book);
        managerFactory.close();
    }


    private static Book getBook(long isbn, String name, String author) {
        Book book = new Book();
        book.setIsbn(isbn);
        book.setName(name);
        book.setAuthor(author);
        return book;
```

```
    }

}
```

Como esperado, toda operação acontece pelo `EntityManager`. A informação só vai definitivamente para o banco de dados quando se realiza o commit da transação. Como? O ponto é que, mesmo o Cassandra não tendo transação de maneira nativa no banco de dados, o Hibernate acaba simulando esse comportamento, que pode ser perigoso, principalmente, em ambientes distribuídos.

O recurso de JPQL, Java Persistence Query Language, é uma query de consulta criada para o JPA e também está disponível dentro do Hibernate OGM, tudo isso graças ao Hibernate Search, que permite a busca por campos além da partition key. Existe a contrapartida: esse campo não poderá será analisado dentro da busca, ou seja, dentro da anotação `Field` o atributo `analizy` precisará ser definido como `Analyze.NO` (verifique como o campo `name` foi anotado dentro da classe).

```
public class App2 {


    public static void main(String[] args) {
        EntityManagerFactory managerFactory =
Persistence.createEntityManagerFactory("hibernate");
        EntityManager manager = managerFactory.createEntityManager();
        manager.getTransaction().begin();
        String name = "Clean Code";
        Book cleanCode = getBook(1L, "Clean Code", "Robert Cecil Martin");
        Book cleanArchitecture = getBook(2L, "Clean Architecture", "Robert
Cecil Martin");
        Book agile = getBook(3L, "Agile Principles, Patterns, and
Practices in C#", "Robert Cecil Martin");
        Book effectiveJava = getBook(4L, "Effective Java", "Joshua
Bloch");
        Book javaConcurrency = getBook(5L, "Java Concurrency", "Robert
Cecil Martin");
```

```
        manager.merge(cleanCode);
        manager.merge(cleanArchitecture);
        manager.merge(agile);
        manager.merge(effectiveJava);
        manager.merge(javaConcurrency);
        manager.getTransaction().commit();

        Query query = manager.createQuery("select b from book b where name
= :name");
        query.setParameter("name", name);
        List<Book> books = query.getResultList();
        System.out.println("books: " + books);
        managerFactory.close();
    }


    private static Book getBook(long isbn, String name, String author) {
        Book book = new Book();
        book.setIsbn(isbn);
        book.setName(name);
        book.setAuthor(author);
        return book;
    }

}
```

Com o motor de busca ativado no projeto, é possível realizar pesquisas utilizando todos os recursos do Lucene. Por exemplo, realizar busca do tipo `term`, que procura uma palavra dentro do texto, além de definir analisadores, tokens etc. Como mostra o código a seguir, o `FullTextEntityManager` é uma especialização do `EntityManager` com recursos do motor de busca. Isso faz com que a busca seja possível para campos que não sejam ID, além de oferecer recursos bem poderosos.

```
public class App3 {


    public static void main(String[] args) {
```

```
        EntityManagerFactory managerFactory =
Persistence.createEntityManagerFactory("hibernate");
        EntityManager manager = managerFactory.createEntityManager();
        manager.getTransaction().begin();
        manager.merge(getBook(1L, "Clean Code", "Robert Cecil Martin"));
        manager.merge(getBook(2L, "Clean Architecture", "Robert Cecil
Martin"));
        manager.merge(getBook(3L, "Agile Principles, Patterns, and
Practices in C#", "Robert Cecil Martin"));
        manager.merge(getBook(4L, "Effective Java", "Joshua Bloch"));
        manager.merge(getBook(5L, "Java Concurrency", "Robert Cecil
Martin"));
        manager.getTransaction().commit();
        FullTextEntityManager fullTextEntityManager =
Search.getFullTextEntityManager(manager);

        QueryBuilder qb = fullTextEntityManager.getSearchFactory()
                .buildQueryBuilder().forEntity(Book.class).get();
        org.apache.lucene.search.Query query = qb
                .keyword()
                .onFields("name", "author")
                .matching("Robert")
                .createQuery();

        Query persistenceQuery =
fullTextEntityManager.createFullTextQuery(query, Book.class);
        List<Book> result = persistenceQuery.getResultList();
        System.out.println(result);

        manager.close();
    }


    private static Book getBook(long isbn, String name, String author) {
        Book book = new Book();
        book.setIsbn(isbn);
        book.setName(name);
        book.setAuthor(author);
        return book;
    }
```

```
}
```

Ainda resta um desafio: uma vez que não conseguimos representar um Set de UDT dentro do JPA, como faremos a busca pela categoria? A resposta surge utilizando os recursos do Hibernate Search. O que faremos é adicionar um novo campo, o `category`. Esse campo será uma `String` e conterá as categorias separadas por vírgula. Depois, todo o trabalho será realizado pelo motor de busca. Com isso, será necessário realizar uma mudança dentro da família de coluna para adicionar o novo campo.

```
ALTER COLUMNFAMILY library.book ADD category text;
```

Com o campo criado, basta alimentar o campo, que o Hibernate Search integrado com o OGM Cassandra se encarregará de fazer o trabalho pesado da indexação do campo e tratá-lo em conjunto com o Apache Lucene.

```
public class App4 {


    public static void main(String[] args) {
        EntityManagerFactory managerFactory =
Persistence.createEntityManagerFactory("hibernate");
        EntityManager manager = managerFactory.createEntityManager();
        manager.getTransaction().begin();
        manager.merge(getBook(1L, "Clean Code", "Robert Cecil Martin",
"Java,OO"));
        manager.merge(getBook(2L, "Clean Architecture", "Robert Cecil
Martin", "Good practice"));
        manager.merge(getBook(3L, "Agile Principles, Patterns, and
Practices in C#", "Robert Cecil Martin", "Good practice"));
        manager.merge(getBook(4L, "Effective Java", "Joshua Bloch", "Java,
Good practice"));
        manager.merge(getBook(5L, "Java Concurrency", "Robert Cecil
Martin", "Java,OO"));
        manager.merge(getBook(6L, "Nosql Distilled", "Martin Fowler",
"Java,OO"));
        manager.getTransaction().commit();
```

```
        FullTextEntityManager fullTextEntityManager =
Search.getFullTextEntityManager(manager);

        QueryBuilder builder =
fullTextEntityManager.getSearchFactory().buildQueryBuilder().forEntity(Boo
k.class).get();
        org.apache.lucene.search.Query luceneQuery =
builder.keyword().onFields("category").matching("Java").createQuery();

        Query query =
fullTextEntityManager.createFullTextQuery(luceneQuery, Book.class);
        List<Book> result = query.getResultList();
        System.out.println(result);
        managerFactory.close();

    }


    private static Book getBook(long isbn, String name, String author,
String category) {
        Book book = new Book();
        book.setIsbn(isbn);
        book.setName(name);
        book.setAuthor(author);
        book.setCategory(category);
        return book;
    }

}
```

Conclusão

Utilizar uma API que o desenvolvedor já conhece para navegar em um novo paradigma no mundo da persistência é uma grande estratégia da qual o Hibernate tira o proveito. Neste capítulo vimos como utilizar a API JPA para trabalhar com o Apache Cassandra, o que diminui e muito a curva de aprendizagem para quem é experiente em Java. Porém, essa facilidade cobra o seu preço, uma vez que o JPA foi feito para o banco de dados relacional. Existem

diversas lacunas que API tende a não cobrir no mundo NoSQL, por exemplo, operações de maneira assíncrona, assim como definições de nível de consistência existente no Cassandra. No próximo capítulo veremos também uma especificação do mundo Java, porém, uma API criada especificamente para a o mundo não relacional.

Capítulo 8
Criando um aplicativo com Java e Spring

O Spring framework é um projeto open source cujo objetivo é facilitar o desenvolvimento Java e hoje se tornou umas das ferramentas mais populares em todo o mundo. A sua história se colide muito com a do Java EE uma vez que ele nasceu para preencher a lacuna que o Java EE não conseguia, além de simplificar diversos pontos, como a parte de segurança. No seu começo, ele era apenas um framework de injeção de dependência, porém, atualmente ele tem diversos subprojetos, dentre os quais podemos citar:

- Spring Batch
- Spring Boot
- Spring security
- Spring LDAP
- Spring XD
- Spring Data
- E muito mais

Neste capítulo, será apresentado um pouco do mundo Spring integrado com o uso do Cassandra.

8.1 Facilitando o acesso aos dados com Spring Data

O Spring Data é um dos vários projetos dentro do guarda-chuva do framework. Este subprojeto tem como maior objetivo facilitar a integração entre Java e os bancos de dados. Existem diversos bancos de dados que são suportados dentro do Spring. Dentre as suas maiores features estão:

- Uma grande abstração *object-mapping*.
- O *query by method*, que se baseia na query dinâmica realizada na interface.
- Fácil integração com outros projetos dentro do Spring, dentre eles, o Spring MVC e o JavaConfig.
- Suporte por auditoria.

Dentro do Spring Data, existe o Spring Data Cassandra, que tem suporte à criação de repositórios, a operações síncronas e assíncronas, a recursos como *query builders* e uma altíssima abstração do Cassandra - a ponto de tornar dispensável aprender o Cassandra Query Language. Para adentrar no maravilhoso mundo do Spring, o primeiro passo é adicionar a dependência no projeto.

```
<dependency>
    <groupId>org.springframework.data</groupId>
    <artifactId>spring-data-cassandra</artifactId>
    <version>2.2.4.RELEASE</version>
</dependency>
```

Definidas as dependências, o próximo passo é o código de infraestrutura que ativa o Spring e a configuração de conexão com o Cassandra. A classe `Config` tem duas anotações: uma para procurar os componentes dentro de um pacote específico e outra para fazer algo similar aos repositórios Cassandra. Já a classe `CassandraConfig` dispõe as configurações para a conexão com o Cassandra, por exemplo, o keyspace a ser utilizado, configurações do cluster e o Mapper Cassandra Spring.

> O Mapper Cassandra Spring, como os Mappers em geral, tem a responsabilidade de fazer conversão de uma entidade de negócio em Java para o Cassandra ou vice-versa.

```
@ComponentScan("com.nosqlxp.cassandra")
@EnableCassandraRepositories("com.nosqlxp.cassandra")
public class Config {
}
```

```
@Configuration
public class CassandraConfig extends AbstractCassandraConfiguration {

    @Override
    protected String getKeyspaceName() {
        return "library";
    }

    @Bean
    public CassandraClusterFactoryBean cluster() {
        CassandraClusterFactoryBean cluster =
                new CassandraClusterFactoryBean();
        cluster.setContactPoints("127.0.0.1");
        cluster.setPort(9042);
        return cluster;
    }

    @Bean
    public CassandraMappingContext cassandraMapping() {
        BasicCassandraMappingContext mappingContext = new
BasicCassandraMappingContext();
        mappingContext.setUserTypeResolver(new
SimpleUserTypeResolver(cluster().getObject(), getKeyspaceName()));
        return mappingContext;
    }


}
```

Para obter mais informações sobre o framework Spring, um bom livro é o *Vire o jogo com Spring* da Casa do Código: https://www.casadocodigo.com.br/products/livro-spring-framework

Código de configuração ou infraestrutura criado, trabalharemos no primeiro exemplo, que é a leitura e a escrita do livro. A modelagem

acontece de maneira simples e intuitiva graças às anotações do Spring Data Cassandra:

- `Table` mapeia a entidade.
- `PrimaryKey` identifica a chave primária dentro da entidade.
- `Column` define os atributos que serão persistidos dentro do Cassandra.

```
@Table
public class Book {

    @PrimaryKey
    private Long isbn;

    @Column
    private String name;

    @Column
    private String author;

    @Column
    private Set<String> categories;

    //getter and setter
}
```

Para a manipulação dos dados, existe a classe `CassandraTemplate`, que é um esqueleto para realizar uma operação entre o Cassandra e o objeto mapeado. Funciona de uma maneira bem análoga ao padrão Template Method que define o esqueleto para do algoritmo, porém, para operações no banco de dados com o Cassandra, mapeando as entidades para o banco e vice-versa. Um ponto importante do `CassandraTemplate` é que é possível realizar chamadas CQL, que ele se encarregará de converter para o objeto alvo - nesse caso, o `Book`.

```
public class App {
    private static final String KEYSPACE = "library";
    private static final String COLUMN_FAMILY = "book";
```

```
    public static void main(String[] args) {

        try (AnnotationConfigApplicationContext ctx = new
AnnotationConfigApplicationContext(Config.class)) {

            CassandraTemplate template =
ctx.getBean(CassandraTemplate.class);

            Book cleanCode = getBook(1L, "Clean Code", "Robert Cecil
Martin", Sets.newHashSet("Java", "OO"));
            Book cleanArchitecture = getBook(2L, "Clean Architecture",
"Robert Cecil Martin", Sets.newHashSet("Good practice"));
            Book effectiveJava = getBook(3L, "Effective Java", "Joshua
Bloch", Sets.newHashSet("Java", "Good practice"));
            Book nosql = getBook(4L, "Nosql Distilled", "Martin Fowler",
Sets.newHashSet("NoSQL", "Good practice"));

            template.insert(cleanCode);
            template.insert(cleanArchitecture);
            template.insert(effectiveJava);
            template.insert(nosql);

            List<Book> books =
template.select(QueryBuilder.select().from(KEYSPACE, COLUMN_FAMILY),
Book.class);
            System.out.println(books);


        }
    }


    private static Book getBook(long isbn, String name, String author,
Set<String> categories) {
        Book book = new Book();
        book.setIsbn(isbn);
        book.setName(name);
        book.setAuthor(author);
        book.setCategories(categories);
        return book;
```

```
    }
}
```

A classe `AnnotationConfigApplicationContext` levanta o contêiner Spring, varrendo as classes anotadas e definidas, em busca das injeções de dependência. Ela permite o uso de *try-resources*, ou seja, tão logo sai do bloco do `try` , a própria JVM se encarregará de chamar e método `close` e fechar o contêiner do Spring para o desenvolvedor.

Para o próximo passo, é possível perceber que não é realizado nenhum contato com o CQL em si, apenas com o template do Cassandra. No código a seguir utilizaremos o `CassandraTemplate` e realizaremos as operações de inserir, recuperar e remover. Apenas para lembrar, não utilizamos o `update` , uma vez que ele funciona como um alias para o `insert` .

```
public class App2 {


    public static void main(String[] args) {

        try (AnnotationConfigApplicationContext ctx = new
AnnotationConfigApplicationContext(Config.class)) {

            CassandraTemplate template =
ctx.getBean(CassandraTemplate.class);

            Book cleanCode = getBook(1L, "Clean Code", "Robert Cecil
Martin", Sets.newHashSet("Java", "OO"));
            Book cleanArchitecture = getBook(2L, "Clean Architecture",
"Robert Cecil Martin", Sets.newHashSet("Good practice"));
            Book effectiveJava = getBook(3L, "Effective Java", "Joshua
Bloch", Sets.newHashSet("Java", "Good practice"));
            Book nosql = getBook(4L, "Nosql Distilled", "Martin Fowler",
Sets.newHashSet("NoSQL", "Good practice"));

            template.insert(cleanCode);
            template.insert(cleanArchitecture);
            template.insert(effectiveJava);
```

```
            template.insert(nosql);

            Book book = template.selectOneById(1L, Book.class);
            System.out.println(book);
            template.deleteById(1L, Book.class);

        }

    }

    private static Book getBook(long isbn, String name, String author,
Set<String> categories) {
        Book book = new Book();
        book.setIsbn(isbn);
        book.setName(name);
        book.setAuthor(author);
        book.setCategories(categories);
        return book;
    }

}
```

Para a última parte do desafio, que consiste na leitura das categorias do livro, as anotações são as mesmas utilizadas no caso do livro, com exceção do UDT `Book`, que possui as anotações `UserDefinedType` e `CassandraType`, definindo o nome do UDT e as informações para o campo, respectivamente.

```
@Table
public class Category {

    @PrimaryKey
    private String name;

    @Column
    private Set<BookType> books;

   //getter and setter
}
```

```
@UserDefinedType("book")
public class BookType {

    @CassandraType(type = DataType.Name.BIGINT)
    private Long isbn;

    @CassandraType(type = DataType.Name.TEXT)
    private String name;

    @CassandraType(type = DataType.Name.TEXT)
    private String author;

    @CassandraType(type = DataType.Name.SET, typeArguments =
DataType.Name.TEXT)
    private Set<String> categories;

    //getter and setter
}
```

Além das anotações do UDT, nada se difere dos dois primeiros caso com relação à consulta pela chave e a persistência do banco de dados. No código a seguir mostraremos uma interação entre as entidades e o tipo UTC. Um aspecto importante é o grande poder que tem o Cassandra e as possiblidades de modelagem sem realizar os famosos *joins* no SQL. Faremos uma inserção da categoria, que tem um `Set` de tipo de livro, e o tipo de livro que é um UDT terá um `Set` de String.

```
public class App3 {

    private static final String KEYSPACE = "library";
    private static final String COLUMN_FAMILY = "category";

    public static void main(String[] args) {

        try (AnnotationConfigApplicationContext ctx = new
AnnotationConfigApplicationContext(Config.class)) {

            CassandraTemplate template =
ctx.getBean(CassandraTemplate.class);
```

```
            BookType cleanCode = getBook(1L, "Clean Code", "Robert Cecil
Martin", Sets.newHashSet("Java", "OO"));
            BookType cleanArchitecture = getBook(2L, "Clean Architecture",
"Robert Cecil Martin", Sets.newHashSet("Good practice"));
            BookType effectiveJava = getBook(3L, "Effective Java", "Joshua
Bloch", Sets.newHashSet("Java", "Good practice"));
            BookType nosqlDistilled = getBook(4L, "Nosql Distilled",
"Martin Fowler", Sets.newHashSet("NoSQL", "Good practice"));


            Category java = getCategory("Java", Sets.newHashSet(cleanCode,
effectiveJava));
            Category oo = getCategory("OO", Sets.newHashSet(cleanCode,
effectiveJava, cleanArchitecture));
            Category goodPractice = getCategory("Good practice",
Sets.newHashSet(cleanCode, effectiveJava, cleanArchitecture,
nosqlDistilled));
            Category nosql = getCategory("NoSQL",
Sets.newHashSet(nosqlDistilled));

            template.insert(java);
            template.insert(oo);
            template.insert(goodPractice);
            template.insert(nosql);

            List<Category> categories =
template.select(QueryBuilder.select().from(KEYSPACE, COLUMN_FAMILY),
Category.class);
            System.out.println(categories);
        }

    }

    private static Category getCategory(String name, Set<BookType> books)
{
        Category category = new Category();
        category.setName(name);
        category.setBooks(books);
        return category;
    }
```

```
    private static BookType getBook(long isbn, String name, String author,
Set<String> categories) {
        BookType book = new BookType();
        book.setIsbn(isbn);
        book.setName(name);
        book.setAuthor(author);
        book.setCategories(categories);
        return book;
    }

}

public class App4 {


    public static void main(String[] args) {

        try (AnnotationConfigApplicationContext ctx = new
AnnotationConfigApplicationContext(Config.class)) {

            CassandraTemplate template =
ctx.getBean(CassandraTemplate.class);

            BookType cleanCode = getBook(1L, "Clean Code", "Robert Cecil
Martin", Sets.newHashSet("Java", "OO"));
            BookType cleanArchitecture = getBook(2L, "Clean Architecture",
"Robert Cecil Martin", Sets.newHashSet("Good practice"));
            BookType effectiveJava = getBook(3L, "Effective Java", "Joshua
Bloch", Sets.newHashSet("Java", "Good practice"));
            BookType nosqlDistilled = getBook(4L, "Nosql Distilled",
"Martin Fowler", Sets.newHashSet("NoSQL", "Good practice"));


            Category java = getCategory("Java", Sets.newHashSet(cleanCode,
effectiveJava));
            Category oo = getCategory("OO", Sets.newHashSet(cleanCode,
effectiveJava, cleanArchitecture));
            Category goodPractice = getCategory("Good practice",
Sets.newHashSet(cleanCode, effectiveJava, cleanArchitecture,
nosqlDistilled));
```

```
            Category nosql = getCategory("NoSQL",
Sets.newHashSet(nosqlDistilled));

            template.insert(java);
            template.insert(oo);
            template.insert(goodPractice);
            template.insert(nosql);

            Category category = template.selectOneById("Java",
Category.class);
            System.out.println(category);
            template.deleteById("Java", Category.class);

        }

    }

    private static Category getCategory(String name, Set<BookType> books)
{
        Category category = new Category();
        category.setName(name);
        category.setBooks(books);
        return category;
    }

    private static BookType getBook(long isbn, String name, String author,
Set<String> categories) {
        BookType book = new BookType();
        book.setIsbn(isbn);
        book.setName(name);
        book.setAuthor(author);
        book.setCategories(categories);
        return book;
    }

}
```

Além da classe template, o Spring Data Cassandra conta com o conceito de **repositórios dinâmicos**, no qual o desenvolvedor cria uma interface e o Spring se responsabilizará da respectiva

implementação. A nova interface herdará de `CassandraRepository`, que já possui um grande número de operações para o banco de dados.

Além disso, é possível utilizar o conceito de `query by method`, com o qual, ao utilizar as conversões de busca no nome do método, o Spring fará todo o trabalho pesado. Com esses repositórios, temos uma abstração valiosa que reduz o número de código, gerando uma altíssima produtividade.

```
@Repository
public interface BookRepository extends CassandraRepository<Book, Long> {

    @Query("select * from book")
    List<Book> findAll();
}


public class App5 {

    public static void main(String[] args) {

        try (AnnotationConfigApplicationContext ctx = new
AnnotationConfigApplicationContext(Config.class)) {

            BookRepository repository = ctx.getBean(BookRepository.class);
            Book cleanCode = getBook(1L, "Clean Code", "Robert Cecil
Martin", Sets.newHashSet("Java", "OO"));
            Book cleanArchitecture = getBook(2L, "Clean Architecture",
"Robert Cecil Martin", Sets.newHashSet("Good practice"));
            Book effectiveJava = getBook(3L, "Effective Java", "Joshua
Bloch", Sets.newHashSet("Java", "Good practice"));
            Book nosql = getBook(4L, "Nosql Distilled", "Martin Fowler",
Sets.newHashSet("NoSQL", "Good practice"));

            repository.insert(cleanCode);
            repository.insert(cleanArchitecture);
            repository.insert(effectiveJava);
            repository.insert(nosql);
```

```
            List<Book> books = repository.findAll();
            System.out.println(books);

            Optional<Book> book = repository.findById(1L);
            System.out.println(book);


        }
    }


    private static Book getBook(long isbn, String name, String author,
Set<String> categories) {
        Book book = new Book();
        book.setIsbn(isbn);
        book.setName(name);
        book.setAuthor(author);
        book.setCategories(categories);
        return book;
    }
}
```

A interface `CassandraRepository` é uma especialização do `CrudRepository` para operações do Cassandra. Já o `CrudRepository` é uma especialização do `Repository`. Essas interfaces fazem parte do Spring Data. Para mais informações: https://docs.spring.io/spring-data/data-commons/docs/2.1.x/reference/html/

O Spring Data Cassandra tem muitos mais recursos, como operações assíncronas que facilitam e muito o dia a dia de quem desenvolve. Para saber mais: https://docs.spring.io/spring-data/cassandra/docs/2.1.2.RELEASE/reference/html/

Conclusão

O Spring framework é um projeto que trouxe uma grande inovação para o mundo Java. Seus recursos e facilitações fazem com que ele

tenha alta popularidade. Dentro da comunicação com o banco de dados, existe o Spring Data com diversas facilitações a ponto de não ser necessário aprender o Cassandra Query Language. Neste capítulo, tivemos uma introdução sobre Spring Data Cassandra e seus recursos. Sua facilidade e a integração com o contêiner do Spring faz com que o Spring Data seja uma ótima solução para aplicativos que já utilizam ou pretendem utilizar o Spring de alguma forma.

Capítulo 9
Criando um aplicativo com Java EE, ops, Jakarta EE

Um dos grandes diferenciais dentro do mundo Java são as especificações. Essas especificações são regidas pelo órgão JCP, cujo foco é garantir uma comunicação transparente, uma forte participação entre os grupos de usuários Java ao redor do mundo.

Além da comunidade, existem diversos benefícios técnicos, por exemplo, a possibilidade de se desfrutar do *multi-vendors* evitando ficar preso a um único fornecedor (o conceito de *vendor lock-in*), compromisso com a retrocompatibilidade, uma rica documentação realizada por diversas empresas, dentre outros.

O objetivo deste capítulo é falar um pouco sobre um dos frutos desse órgão: o Java EE e sua solução para o mundo não relacional.

9.1 O que é Jakarta EE?

Uma das grandes mudanças no mundo Java EE é que, em 2017, ele foi doado para a Eclipse Foundation pela Oracle, então ali se iniciou o processo de transição. Assim, a Oracle não é mais responsável pela especificação Java EE, todo esse trabalho será mantido pela Foundation. Um ponto importante a salientar é o que foi transferido:

- Código que pertence às APIs;
- As documentações;
- As implementações de referências que pertenciam à Oracle (vale lembrar de que existem especificações que não são gerenciadas pela Oracle, como é o caso do Bean Validation e o CDI).

Porém, não foi entregue o direito do nome Java, dessa forma, não era possível continuar se chamando “Java EE”. Com isso, foi realizado uma votação por dentro da comunidade e eles elegeram o novo nome: Jakarta EE, além do novo logo. Assim, de uma maneira geral o Jakarta EE é apenas a nova casa do Java EE.

Como premissa, sob nova direção com Eclipse Foundation, o Jakarta EE manterá compatibilidade com a última versão do Java EE, a versão 8, além de trazer novidades para dentro da plataforma.

Um dos pontos importantes é que para trazer mais novidades para a plataforma está sendo criado um novo ciclo para especificação no lugar das famosas JSRs, com foco principal de facilitar o desenvolvimento, realizar entregas rápidas e receber feedbacks de maneira mais rápida da comunidade.

Como primeira especificação, nasceu o Eclipse JNoSQL, cujo foco é realizar integração entre os bancos NoSQL e Java.

9.2 Utilizando Jakarta NoSQL, a primeira especificação do Jakarta EE

O Jakarta NoSQL é um framework que realiza a integração entre as aplicações Java com bancos de dados NoSQL.

Ele define um grupo de APIs cujo objetivo é padronizar a comunicação entre a maioria dos bancos de dados e suas operações comuns. Isso ajuda a diminuir o acoplamento com este tipo de tecnologia utilizada nas aplicações atuais.

O projeto tem duas camadas:

1. Camada de comunicação: é um grupo de APIs que define a comunicação com os bancos de dados não relacionais. Comparado aos tradicionais bancos não relacionais, eles são semelhantes às APIs JDBC. Ela contém quatro módulos, um para cada tipo de banco NoSQL: chave-valor, família de coluna, documento e grafos.

2. Camada de mapeamento: API que ajuda o desenvolvedor na integração com o banco não relacional, sendo orientada a anotações e utilizando tecnologias como injeção de dependências e Bean Validation, o que torna simples para os desenvolvedores a utilizarem. Comparando com os clássicos RDBMS, essa camada pode ser comparada com o JPA ou outros frameworks de mapeamentos como o Hibernate.

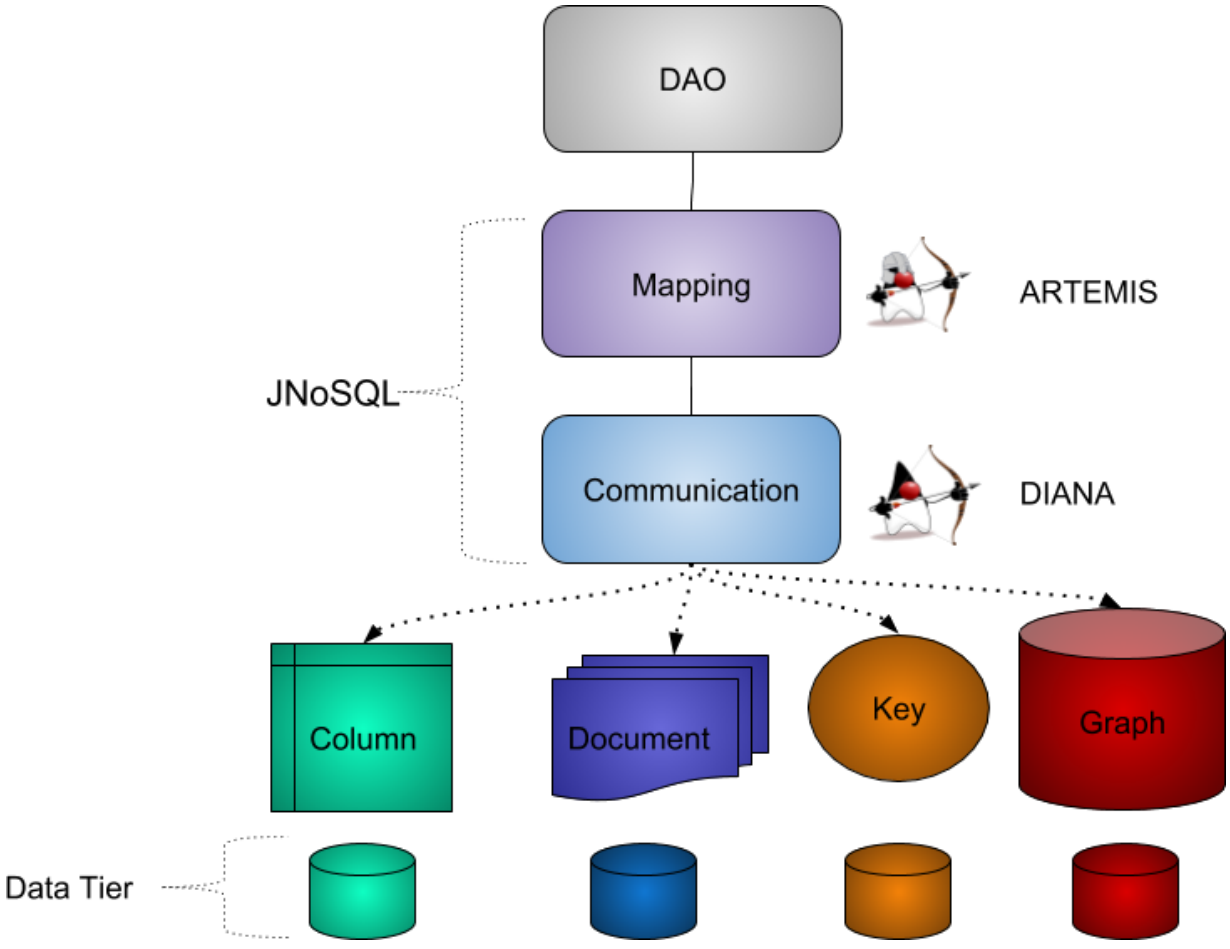

Figura 9.1: Arquitetura do Eclipse JNoSQL

Assim como no Spring Data, o Jakarta NoSQL trabalha em conjunto com um motor de injeção de dependência, porém, no projeto se utiliza a especificação do CDI. Utilizando o mesmo princípio, começaremos com os códigos e arquivos de configuração que são necessários para fazer o contêiner do CDI levantar, além de fazer da comunicação com o Cassandra.

> O CDI é um framework de injeção de dependência muito interessante e com diversos recursos, como definição de escopo, disparo de evento de maneira síncrona e assíncrona além de ser o padrão do mundo Java. Para conhecer mais uma excelente leitura é o livro de CDI da Casa do Código: https://www.casadocodigo.com.br/products/livro-cdi

O arquivo de configuração é o `bean.xml` , que fica localizado dentro do `META-INF` (a mesma localização do `persistence.xml` do JPA). Também existirá o `CassandraProducer` , que terá como responsabilidade criar a conexão com o Cassandra.

```
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
        http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd"
       bean-discovery-mode="all">
</beans>
```

A classe de configuração “ensina” ao CDI como gerar a dependência de `CassandraColumnFamilyManager` , que é a classe responsável por realizar a comunicação entre o Java e o banco de dados. A classe `Settings` representa as informações para conexão com o banco de dados. Nela, é possível, por exemplo, definir a senha, usuário, clusters, dentre outras informações.

```
@ApplicationScoped
public class CassandraProducer {

    private static final String KEYSPACE = "library";
```

```
    private CassandraConfiguration cassandraConfiguration;

    private CassandraColumnFamilyManagerFactory managerFactory;

    @PostConstruct
    public void init() {
        cassandraConfiguration = new CassandraConfiguration();
        Settings settings =
Settings.of(Collections.singletonMap("cassandra-host-1", "localhost"));
        managerFactory = cassandraConfiguration.get(settings);
    }


    @Produces
    @ApplicationScoped
    public CassandraColumnFamilyManager getManagerCassandra() {
        return managerFactory.get(KEYSPACE);
    }

    public void dispose(@Disposes CassandraColumnFamilyManager manager) {
        manager.close();
        managerFactory.close();
    }

}
```

O próximo passo se encontra em realizar a modelagem da entidade de livros. Caso você venha do mundo JPA verá que os conceitos são os mesmos, ou seja, usamos a anotação `Entity` para mapear uma entidade, a anotação `Id` para identificar o atributo que será o identificador único, além da anotação `Column` para identificar os outros campos que serão persistidos.

```
@Entity("book")
public class Book {

    @Id("isbn")
    private Long isbn;
```

```
    @Column
    private String name;

    @Column
    private String author;

    @Column
    private Set<String> categories;

    //getter and setter
```

O Jakarta NoSQL tem como sua maior característica a integração com um framework de injeção de dependência, assim como o Spring Data, porém, a diferença é que a especificação utiliza o CDI, que é a especificação do mundo Java.

Uma outra semelhança entre as ferramentas de integração se dá pelo fato de que o primeiro passo é levantar o contêiner. O projeto possui uma classe template para operações dentro do Mapper: o `ColumnTemplate` , porém, ele funciona como esqueleto de operações de um Mapper para todos os bancos do tipo família de coluna, ou seja, seria possível trocar para o Hbase com pouco ou nenhum impacto no código.

```
public class App
{
    public static void main( String[] args )
    {
        try(SeContainer container =
SeContainerInitializer.newInstance().initialize()) {
            ColumnTemplate template =
container.select(ColumnTemplate.class).get();

            Book cleanCode = getBook(1L, "Clean Code", "Robert Cecil
Martin", Sets.newHashSet("Java", "OO"));
            Book cleanArchitecture = getBook(2L, "Clean Architecture",
"Robert Cecil Martin", Sets.newHashSet("Good practice"));
            Book effectiveJava = getBook(3L, "Effective Java", "Joshua
Bloch", Sets.newHashSet("Java", "Good practice"));
            Book nosql = getBook(4L, "Nosql Distilled", "Martin Fowler",
```

```
Sets.newHashSet("NoSQL", "Good practice"));

            template.insert(cleanCode);
            template.insert(cleanArchitecture);
            template.insert(effectiveJava);
            template.insert(nosql);

            ColumnQuery query = select().from("book").build();
            Stream<Book> books = template.select(query);
            books.forEach(System.out::println);
        }
    }

    private static Book getBook(long isbn, String name, String author,
Set<String> categories) {
        Book book = new Book();
        book.setIsbn(isbn);
        book.setName(name);
        book.setAuthor(author);
        book.setCategories(categories);
        return book;
    }
}
```

A API do Jakarta NoSQL possui recursos interessantes como a query e a API fluente. Vale salientar que esses recursos são portáveis para outros bancos de dados não relacionais do tipo família de coluna como o Apache Hbase. No código a seguir, mostraremos consultas utilizando a API do Jakarta NoSQL utilizando tanto o recurso de API fluente como utilizando uma query como texto. Um ponto importante é o uso do `Optional` quando se faz busca com apenas um único elemento, ou seja, a API já nasceu acima do Java 8.

```
public class App2 {


    public static void main(String[] args) {

        try(SeContainer container =
```

```
SeContainerInitializer.newInstance().initialize()) {
            ColumnTemplate template =
container.select(ColumnTemplate.class).get();

            Book cleanCode = getBook(1L, "Clean Code", "Robert Cecil
Martin", Sets.newHashSet("Java", "OO"));
            Book cleanArchitecture = getBook(2L, "Clean Architecture",
"Robert Cecil Martin", Sets.newHashSet("Good practice"));
            Book effectiveJava = getBook(3L, "Effective Java", "Joshua
Bloch", Sets.newHashSet("Java", "Good practice"));
            Book nosql = getBook(4L, "Nosql Distilled", "Martin Fowler",
Sets.newHashSet("NoSQL", "Good practice"));

            template.insert(cleanCode);
            template.insert(cleanArchitecture);
            template.insert(effectiveJava);
            template.insert(nosql);


            Optional<Book> book = template.find(Book.class, 1L);
            System.out.println("Book found: " + book);

            template.delete(Book.class, 1L);

            System.out.println("Book found: " + template.find(Book.class,
1L));


            PreparedStatement prepare = template.prepare("select * from
Book where isbn = @isbn");
            prepare.bind("isbn",2L);
            Optional<Book> result = prepare.getSingleResult();
            System.out.println("prepare: " + result);
        }

    }

    private static Book getBook(long isbn, String name, String author,
Set<String> categories) {
        Book book = new Book();
        book.setIsbn(isbn);
```

```
        book.setName(name);
        book.setAuthor(author);
        book.setCategories(categories);
        return book;
    }

}
```

No mapeamento da categoria, a sequência continua semelhante, com diferença do mapeamento do `UDT` . Dentro do Jakarta NoSQL, existe o recurso de extensões que permite APIs de usos específicos para cada banco de dados. Por exemplo, utilizaremos uma extensão do Cassandra que permite que utilizemos a anotação `UDT` para mapear nosso código para o tipo UDT, como mostra o mapeamento a seguir.

```
@Entity("category")
public class Category {

    @Id("name")
    private String name;

    @Column
    @UDT("book")
    private Set<BookType> books;
    //getter and setter
}


public class BookType {

    @Column
    private Long isbn;

    @Column
    private String name;

    @Column
    private String author;
```

```
    @Column
    private Set<String> categories;
    //getter and setter
}
```

Uma vez utilizando anotações específicas do Cassandra, será utilizado o `CassandraTemplate`, que é uma especialização do `ColumnTemplate` com recursos específicos para o Cassandra como, por exemplo, a possibilidade de definir o nível de consistência durante a requisição e realizar queries nativas, ou seja, CQL.

```
public class App4 {


    public static void main(String[] args) {
        try(SeContainer container =
SeContainerInitializer.newInstance().initialize()) {
            CassandraTemplate template =
container.select(CassandraTemplate.class).get();

            BookType cleanCode = getBook(1L, "Clean Code", "Robert Cecil
Martin", Sets.newHashSet("Java", "OO"));
            BookType cleanArchitecture = getBook(2L, "Clean Architecture",
"Robert Cecil Martin", Sets.newHashSet("Good practice"));
            BookType effectiveJava = getBook(3L, "Effective Java", "Joshua
Bloch", Sets.newHashSet("Java", "Good practice"));
            BookType nosqlDistilled = getBook(4L, "Nosql Distilled",
"Martin Fowler", Sets.newHashSet("NoSQL", "Good practice"));


            Category java = getCategory("Java", Sets.newHashSet(cleanCode,
effectiveJava));
            Category oo = getCategory("OO", Sets.newHashSet(cleanCode,
effectiveJava, cleanArchitecture));
            Category goodPractice = getCategory("Good practice",
Sets.newHashSet(cleanCode, effectiveJava, cleanArchitecture,
nosqlDistilled));
            Category nosql = getCategory("NoSQL",
Sets.newHashSet(nosqlDistilled));
```

```
            template.insert(java);
            template.insert(oo);
            template.insert(goodPractice);
            template.insert(nosql);

            Optional<Category> category = template.find(Category.class,
"Java");
            System.out.println(category);
            template.delete(Category.class, "Java");

            PreparedStatement prepare = template.prepare("select * from
Category where name = @name");
            prepare.bind("name","NoSQL");
            Optional<Book> result = prepare.getSingleResult();
            System.out.println("prepare: " + result);
        }

    }

    private static Category getCategory(String name, Set<BookType> books)
{
        Category category = new Category();
        category.setName(name);
        category.setBooks(books);
        return category;
    }

    private static BookType getBook(long isbn, String name, String author,
Set<String> categories) {
        BookType book = new BookType();
        book.setIsbn(isbn);
        book.setName(name);
        book.setAuthor(author);
        book.setCategories(categories);
        return book;
    }

}
```

Além das classes templates, o Jakarta NoSQL oferece suporte ao conceito de interfaces repositórios que segue o mesmo princípio do

Spring Data: interfaces que visam ter um algo grau de abstração para realizar consultas dentro do banco. Ele também traz uma interface que já possui diversos métodos e `method by query` que serão implementados de maneira automática pelo framework. Nesse caso, será utilizado o `CassandraRepository`, que é uma especialização do `Repository` que permite, por exemplo, o uso da anotação `CQL` que, basicamente, executa Cassandra Query Language e converte para a entidade automaticamente.

```
public interface BookRepository extends CassandraRepository<Book, Long> {

    Stream<Book> findAll();

    @CQL("select * from book")
    Stream<Book> findAll1();

    @Query("select * from Book")
    Stream<Book> findAll2();
}
```

> A diferença entre as anotações `CQL` e `Query` é que a primeira executa o CQL, que é a query nativa do Cassandra, assim, exclusiva do framework, e a segunda é a API do Jakarta NoSQL, ou seja, poderá ser executada por outros bancos de dados que suporte a camada de comunicação do projeto de especificação do mundo Jakarta EE.

No código a seguir faremos a nossa integração com o `BookRepository`. O que mais impressiona, certamente, é a facilidade em utilizá-lo uma vez que não precisamos nos preocupar com a implementação da interface. Um ponto importante é que essa interface herda de `CassandraRepository`, o que faz com que esse repositório herde vários métodos como o `save`, `findById`, dentre outros.

```
public class App5
{
```

```
    public static void main( String[] args )
    {
        try(SeContainer container =
SeContainerInitializer.newInstance().initialize()) {
            BookRepository repository =
container.select(BookRepository.class).get();

            Book cleanCode = getBook(1L, "Clean Code", "Robert Cecil
Martin", Sets.newHashSet("Java", "OO"));
            Book cleanArchitecture = getBook(2L, "Clean Architecture",
"Robert Cecil Martin", Sets.newHashSet("Good practice"));
            Book effectiveJava = getBook(3L, "Effective Java", "Joshua
Bloch", Sets.newHashSet("Java", "Good practice"));
            Book nosql = getBook(4L, "Nosql Distilled", "Martin Fowler",
Sets.newHashSet("NoSQL", "Good practice"));

            repository.save(cleanCode);
            repository.save(cleanArchitecture);
            repository.save(effectiveJava);
            repository.save(nosql);

            Optional<Book> book = repository.findById(1L);
            System.out.println(book);

            repository.deleteById(1L);

            System.out.println("Using method query");
            repository.findAll().forEach(System.out::println);
            System.out.println("Using CQL");
            repository.findAll1().forEach(System.out::println);
            System.out.println("Using query JNoSQL");
            repository.findAll2().forEach(System.out::println);

        }
    }

    private static Book getBook(long isbn, String name, String author,
Set<String> categories) {
        Book book = new Book();
        book.setIsbn(isbn);
        book.setName(name);
```

```
        book.setAuthor(author);
        book.setCategories(categories);
        return book;
    }
}
```

O projeto ainda tem diversos recursos que não foram exibidos aqui, por exemplo, a realizar operações de maneira assíncrona. Para saber mais acesse: http://www.jnosql.org/

O código com todo exemplo se encontra em: https://github.com/otaviojava/cassandra-java-code para criar as estruturas do Cassandra, consulte o capítulo “Realizando integração com Java”.

Conclusão

Com uma nova casa e de forma ainda mais vibrante, nasce o projeto Eclipse JNoSQL sob nova direção do Jakarta EE com a Eclipse Foundation. O Jakarta NoSQL tem como propósito facilitar a integração entre Java e NoSQL com a estratégia de dividir a camada de comunicação e mapeamento. Atualmente, ele suporta mais de trinta bancos de dados. Muitas melhorias são esperadas, porém, o grande benefício da plataforma é que ela é totalmente orientada à comunidade. Você mesmo pode sair da cadeira e ajudar o Jakarta EE agora mesmo.

Capítulo 10
Considerações finais

Pronto, após todos esses capítulos estamos chegando ao final do livro. O objetivo desta conclusão da série é mostrar as últimas dicas e recomendações para que você esteja livre para desbravar os mares do Cassandra e desenvolver suas próprias experiências.

10.1 Motor de busca

Como já foi discutido no capítulo de modelagem, no mundo ideal todas as queries deverão ser realizadas a partir da partition key, ou seja, além da desnormalização, deve-se evitar a todo custo a utilização dos recursos como `ALLOW INDEXING` ou índice secundário, do contrário, o Cassandra terá impacto em performance.

Essa duplicação de dados terá a vantagem de performance no momento da leitura, mas também terá o impacto do gerenciamento das informações.

O motor de buscas contém diversas vantagens com relação ao Cassandra. A primeira delas é o fato de conseguir realizar uma busca em outros campos que não seja o partition key. O segundo ponto é que a precisão com motor de busca trará grandes benefícios para o produto também.

Imagine que dentro do livro agora se tenha um resumo, porém, o texto está em HTML. Dentro de um motor de busca, a informação quando inserida é tratada para que, quando a busca for realizada, ela seja otimizada. Por exemplo, levando em consideração o processo do Elasticsearch teremos os seguintes passos:

- **Character Filters**: é o primeiro contato da informação. Ele receberá um stream de caracteres e retornará um stream de caracteres. No nosso exemplo, o formato html será removido de forma que o texto ficará da seguinte forma: `Conheça o Clean Code, o livro de boas práticas de programação .`
- **Tokenizer**: o processo de tokenização em que, dado uma stream de caracteres, ele converte em um token, que geralmente resulta em uma palavra. Por exemplo, a descrição do livro se transformaria em tokens como `["Conheça","o", "Clean","Code","o","livro","de","boas","práticas","de","programação"]` .
- **Token Filter**: nessa etapa, dada uma sequência de tokens, ele será responsável por remover (remover palavras comuns, as *stop words*, como: "de", "o"), adicionar (adicionar sinônimos: programação e software) ou modificar tokens (converter as palestras para minúsculo ou remover acentos das palavras).

De modo que, quando for pesquisar, por exemplo, pela palavra "programação" dentro do motor de busca a informação já estará pré-processada como, então não terá de varrer o texto em busca da informação em cada linha. Além do novo horizonte que o motor de busca permite, é possível adicionar sinônimos, adicionar pesos em campos de buscas (caso a palavra esteja no título, ele ter mais relevância do que quando estiver na descrição) e levar em consideração possíveis erros ortográficos que podem realizados pelo usuário de maneira acidental etc.

No mundo do motor de busca existem algumas opções:

- Apache Lucene: quando mencionamos o Hibernate Search foi mencionado um pouco sobre esse projeto. Ele é open source e se encontra dentro da Apache Foundation, sendo uma API de busca e de indexação de documentos. Dentre os grandes cases de experiência do projeto se encontra o Wikipedia.
- Apache Solr: ou simplesmente Solr é também um projeto open source mantido pela Apache Foudation. Dentre as principais features podemos citar pesquisa de texto, indexação em tempo

de execução e a possiblidade de executar em Clusters. Uma curiosidade é que o Apache Solr é desenvolvido em cima do Apache Lucene.
- Elasticsearch: o Elasticsearch, assim como o Apache Solr, é um motor de busca baseado no Apache Lucene, porém, não é da Apache Foundation (apesar de a licença ser Apache 2.0). Também tem o recurso de trabalhar o motor de busca como cluster e atualmente é considerado o motor de busca mais popular e utilizado do mundo.

Existem alguns projetos, inclusive, que visam justamente à integração entre o motor de busca com o Cassandra são os casos do Solandra (integração do Cassandra com Solr) e o Elassandra (integração do Cassandra com Elasticsearch).

10.2 Validando os dados com Bean Validation

Em uma aplicação é muito comum que alguns campos tenham uma validação para garantir a consistência da aplicação. Por exemplo, os campos de e-mails. O Cassandra não possui nenhum recurso de validação, e para preencher essa lacuna uma opção é uso do Bean Validation, que permite colocar constraints em objetos Java a partir de anotações.

O Bean Validation possui um vasto número de anotações nativa que permitem um alto grau de validações, como tamanho mínimo ou tamanho máximo de um text, validação de e-mail, campo obrigatório, dentre outros.

```
public class User {
@NotNull
@Email
private String email;
}
```

Além de ter várias opções de validação, também é possível criar um novo, de modo que o Bean Validation é uma grande opção caso os dados precisem ter algum tipo de validação para a consistência dos dados.

10.3 Realizando testes de integração com o Cassandra

No mundo da engenharia de software, os testes são uma peça fundamental para a construção de qualquer programa. É a partir deles que é possível garantir que o código atual se encontra estável, que se teste de maneira rápida e sem a interferência humana, além de que tenha maior segurança para refatorar e adicionar novos recursos dentro do sistema. Os testes de integração são um dos tipos de testes de um sistema com os quais se verifica um grupo de componente, por exemplo: a comunicação entre o código Java e o Cassandra. Nesse tópico citarei dois bons exemplos para realizar os testes.

O primeiro deles é o Cassandra Unit, que se comporta de maneira similar ao DBUnit, porém, em vez de um banco relacional, utiliza o Cassandra. Ele levantará um Cassandra embarcado pronto para uso nos seus testes.

Uma grande vantagem de usar tecnologias de build é que nós não precisamos nos preocupar com as dependências. Por exemplo, é possível utilizar o Cassandra Unit integrado com Junit 5.

```
<dependency>
    <groupId>org.cassandraunit</groupId>
    <artifactId>cassandra-unit</artifactId>
    <version>3.3.0.2</version>
    <scope>test</scope>
</dependency>
```

É possível perceber que nesse pequeno teste só foi necessário adicionar um método que levanta uma instância embarcada do Cassandra e outra que limpa.

```
public class SampleCassandraTest {


    @BeforeAll
    public static void before() throws InterruptedException, IOException,
TTransportException {
        EmbeddedCassandraServerHelper.startEmbeddedCassandra();
    }

    @AfterAll
    public static void end() {
        EmbeddedCassandraServerHelper.cleanEmbeddedCassandra();
    }

    @Test
    public void test(){
        Cluster.Builder builder = Cluster.builder();
        builder.addContactPoint("localhost").withPort(9142);
        Cluster cluster = builder.build();
        Session session = cluster.newSession();
        ResultSet resultSet = session.execute("SELECT * FROM
system_schema.keyspaces;");
        Assertions.assertNotNull(resultSet);
        cluster.close();
    }

}
```

`Testcontainers` é uma biblioteca Java que permite a integração do Junit com qualquer imagem docker. Assim, é possível executar o Cassandra através de uma imagem docker. A dependência é algo realmente simples de ser adicionado. Após isso, a diferença com o teste de integração anterior está na dependência e no código de infraestrutura para levantar o Cassandra, desta vez, via contêiner.

```
<dependency>
    <groupId>org.testcontainers</groupId>
    <artifactId>testcontainers</artifactId>
    <version>1.9.1</version>
    <scope>test</scope>
</dependency>
```

Com a dependência definida o código se mostra bem simples. A classe `GenericContainer` representa um contêiner docker. No seu construtor, há uma `String` que é o nome da imagem que o `testcontainer` buscará no dockerhub. Feito isso, o próximo passo foi expor a porta de conexão com o banco de dados, nesse caso a port `9072` . Como todo teste de integração, queremos utilizar o contêiner tão logo ele esteja disponível, e é por esse motivo que setamos a estratégia padrão que espera o novo contêiner executar a próxima linha apenas quando o banco de dados instanciado esteja de pé. Com o banco de dados pronto, o próximo passo é a realizar o teste necessário.

```
public class SampleCassandraContainerTest {


    @Test
    public void test(){

        GenericContainer cassandra =
                new GenericContainer("cassandra")
                        .withExposedPorts(9042)
                        .waitingFor(Wait.defaultWaitStrategy());

        cassandra.start();
        Cluster.Builder builder = Cluster.builder();

builder.addContactPoint(cassandra.getIpAddress()).withPort(cassandra.getFi
rstMappedPort());
        Cluster cluster = builder.build();
        Session session = cluster.newSession();
        ResultSet resultSet = session.execute("SELECT * FROM
system_schema.keyspaces;");
        Assertions.assertNotNull(resultSet);
```

```
        cluster.close();
    }
}
```

10.4 Experimentando outros sabores de Cassandra

Um ponto importante é que existem bancos de dados que competem com o Cassandra seja no mesmo tipo de banco de dados seja para resolver, praticamente, os mesmos problemas. É muito importante que o leitor também conheça esses bancos de dados:

- O DataStax DSE: é uma versão corporativa, fechada e paga fornecida pela DataStax. Ele suporta tudo aquilo que o Cassandra faz atualmente e adiciona novos recursos como analytics e motor de busca já integrado. Um outro ponto interessante é que o DSE é um banco de dados *multi-model*, em outras palavras, é um banco de dados NoSQL que suporta mais de um tipo de banco NoSQL (chave-valor, família de coluna e grafos).
- ScillaDB: é um banco de dados cujo foco é a possibilidade de manter total compatibilidade com o Cassandra, porém, com uma performance extremamente superior. Em teoria, é possível pegar uma aplicação em Cassandra e mudar para esse banco de dados sem impacto algum. Ele também tem imagens oficiais dentro do dockerhub. Uma vez que o leitor entrou no mundo Cassandra vale a tentativa.

Conclusão

Com isso, foram apresentados os conselhos finais e últimas dicas para prosseguir com o Cassandra. Recursos como motor de busca, validação de dados com bean validation e ferramentas para testes de integração são recursos valiosos nos quais é recomendável que

você vá muito mais fundo. O objetivo deste capítulo foi de apenas despertar a curiosidade para continuar evoluindo o Cassandra, integrando com outras ferramentas. Espero que este livro tenha contribuído para mostrar como o Cassandra é simples de uso e está a menos de um passo de distância para integrá-lo com aplicações Java.